TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

RIKU NIEMINEN

CLIENT-SIDE WEB APPLICATION MEMORY MANAGEMENT

Master of Science thesis

Examiner: Prof. Tommi Mikkonen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 9th December 2015

ABSTRACT

RIKU NIEMINEN: Client-Side Web Application Memory Management
Tampere University of Technology
Master of Science thesis, 55 pages
December 2015
Master's Degree Programme in Computer Technology
Major: Software Engineering
Examiner: Prof. Tommi Mikkonen
Keywords: Client-Side Web Application, Memory Management, Single Page Application, SPA, JavaScript, Diagnostics

Today web browsers are used more and more as application runtime environment in addition to their use and origins as document viewers. At the same time web application's architecture is undergoing changes. For instance functionality is being moved from the backend into the client, following the so-called Thick client architecture.

Currently it is quite easy to create client side web applications that do not manage their memory allocations. There has not been large focus in client side application's memory usage for various reasons. However, currently client side web applications are widely being built and some of these applications are expected to be run for extended periods. Longevity of the application requires application's succesful memory management. From the performance point of view it is also beneficial that the application manages its memory succesfully. The client-side behaviour of the application is developed with JavaScript, which has automatically managed memory allocations. However, like all abstractions, automatically managed memory is a leaky abstraction to an undecidable problem.

In this thesis we aim at finding out what it takes to create client side applications that succesfully manage their memory allocations. We will take a look at the tools available for investigating memory issues during application development. We also developed a memory diagnostics module, in order to be able to diagnose application instance's memory usage during its use.

The diagnostics module developed during this thesis was used succesfully to monitor application's memory usage over time. With the use of the data provided by the diagnostics module, we were able to identify memory issues from our demo application. However, currently the Web platform does not enable the creation of cross-browser standard relying solution for diagnosing web application's memory usage.

TIIVISTELMÄ

RIKU NIEMINEN: Web Sovelluksen Asiakaspuolen Muistinkulutuksen Hallinta
Tampereen teknillinen yliopisto
Diplomityö, 55 sivua
Joulukuu 2015
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen
Avainsanat: Web Sovellus, Muistinhallinta, Diagnostiikka, Yhden Sivun Web-Sovellus

Web-selainten käyttö on viime vuosina laajentunut hyperlinkkejä sisältävien dokumenttien esityksestä sovellusten suoritukseen. Samanaikaisesti web-sovellusten arkkitehtuuri on muutoksen alla: entistä enemmän toiminnallisuutta siirtyy palvelimelta selaimeen, mukaillen niin kutsuttua paksu asiakas (Thick client) -arkkitehtuuria.

Tällä hetkellä muistivuotoja sisältävien web-sovellusten kehittäminen on melko helppoa, johtuen osittain siitä, että web-selaimessa suoritettavien sovellusten muistinhallintaan ei ole toistaiseksi kiinnitetty laajalti huomiota. Tilanne on kuitenkin muuttumassa, sillä nykyisin web-selaimessa suoritettavia sovelluksia kehitetään laajalti, ja osan näistä sovelluksista odotetaan olevan käytössä pitkiä aikoja. Sovellusten pitkäikäisyys edellyttää sovelluksen onnistunutta muistinhallintaa. Onnistunut muistinhallinta myös mahdollistaa suorituskykyisten sovellusten kehittämisen.

Tässä työssä tutkitaan web-teknologioiden soveltuvuutta missiokriittisten, robustien ja pitkään ajossa olevien web-sovellusten kehittämiseen. Tarkastelemme käytettävissä olevia työkaluja muistiongelmien kehitysaikaiseen tutkimiseen. Kehitimme myös diagnostiikka-modulin, jonka avulla sovellusinstanssin muistinkäyttöä on mahdollista tarkastella ajonaikaisesti.

Diagnostiikka-modulia käytettiin työn aikana onnistuneesti tunnistamaan sovelluksen muistiinkäyttöön liittyviä ongelmia. Työn aikana kehitetty ratkaisu ei kuitenkaan valitettavasti ole standardien mukainen, mistä johtuen diagnostiikka toimii tällä hetkellä ainoastaan Google Chromessa.

PREFACE

This thesis was made for Valmet's technology mapping needs. During the work on this thesis I have been able to broaden my knowledge on the technologies that will be important in the future of computing. At the same time gaining hands on experience with the technologies, on which I am very grateful of.

Thanks to Valmet, Perttu Kotiluoto and Janne Kytölä for letting me work on such an interesting and current problem. It has been really inspiring to be surrounded by professionals from such multiple fields. I would also like to thank my professor Tommi Mikkonen for guidance, and for the previous research on using the browser as an application platform. In addition I would like to thank Saara for support and proofreading.

Tampere, 15.11.2015

Riku Nieminen

TABLE OF CONTENTS

1. INTRODUCTION

In the last two decades we have experienced the rise of the *World Wide Web* (WWW). The Web (as it is usually called) is an information sharing platform running on top of the *Internet*. The Web is the most powerful information distribution environment in the history of humankind. It has evolved from a system that was used to share documents containing hyperlinks to other documents, into a general purpose application and content distribution environment. [49]

We are still in the very early days of the Web, yet it has already become part of our every day lives. Today web browsers are probably the most widely used applications on desktop computers. In addition to this, browsers are run on diverse computing platforms: PCs, Mobile Devices (phones, tablets), Game Stations and TVs among others. Browsers are under continuous change and their role has been evolving from a document viewer into a full runtime environment. More and more tasks are shifting from the operating system into the browser and from the eyes of the average computer user the browser will effectively become the de facto operating system. [48]

Because the web browser provides such an accessible and ubiquitous platform, it is also starting to gain attention from mission critical application developers. However, using web browsers as an application runtime environment is quite a recent turnup, thus leaving more to be desired. Currently it is very easy to create web applications that contain memory leaks and quite difficult to diagnose such problems. This makes it difficult to create mission critical web applications. The *ECMAScript* standard does not define any interface for the developer to diagnose runtimes memory usage, nor interact with the implementation's garbage collector. This makes it impossible for the application developers to diagnose the application's memory usage with standardized cross-browser solution, in order to identify memory leaks from live applications.

In this thesis we focus on using the browser as an application platform for making performant and mission critical single page applications. The web applications we are interested in are expected to be running for extended periods of time, where

the use is measured in months rather than minutes. In this use case the role of succesful memory management becomes compulsory. These requirements have lead us to further narrow our focus on web application's memory management.

We will look into the current status and future of web application development and web applications memory management. During the work on this thesis we have implemented a memory diagnostics module that can be used to diagnose web applications memory consumption. During this thesis we implemented a demo application that uses the aforementioned module to diagnose the applications memory consumption in a real environment. Chapter 2 focuses on using the web browser as an application platform. In chapter 3 we look at JavaScript's memory management and Garbage Collection. Chapter 4 focuses on creating memory leak free web applications, detecting memory leaks, and diagnosing memory consumption during runtime. In chapter 5 we evaluate the memory diagnostics module's usefulness, the status of the web platform and tools, and look into possible ways for improving these. Chapter 6 provides a summary in which we summarize the findings made during the work on this thesis.

2. WEB APPLICATIONS

2.1 Background

During the last two decades it has been quite overlooked for web developers to have an in depth understanding of underlying web technologies. Web developers and designers have been relying heavily on tools to abstract away the details of making web applications. While the use of tools has made many current web applications possible, understanding of the underlying concepts should not be overlooked. Heavy reliance on tools has been able to hide the gruesome details and for a long while not enough pressure was put into improving the technological foundation [24]. However, in the past few years web technologies have matured remarkably: the current advances include standards like, *HTML5*, *CSS3* and *ECMAScript edition 6*, which enable the development of more desktop-like applications, among other things.

When creating web software the division of work has been aligned with the architecture of the applications. The architecture used to build traditional web applications has been three-tier architecture. Three-tier architecture divides the application into three parts: presentation logic, business logic and data logic (Figure 2.1). The presentation layer consists of a user interface (UI) and it also controls the user's interactions. The business logic contains the application's business logic and the data tier provides access to data. In these architectures the presentation layer has resided in the browser (user-agent, client) and the business logic and data tier have resided in the server. [44]

When developing the above-mentioned systems, the work has been roughly divided, so that web designers work on the front-end and web developers and database experts work on the back end (server-side) [59]. The architecture and division of work partly explains why there have been so many technologies, paradigms and programming languages involved when creating web software, which again is part of the reason why the portability of development experience has been quite poor. The division of work has also influenced the dependence on tools especially in the presentation layer.

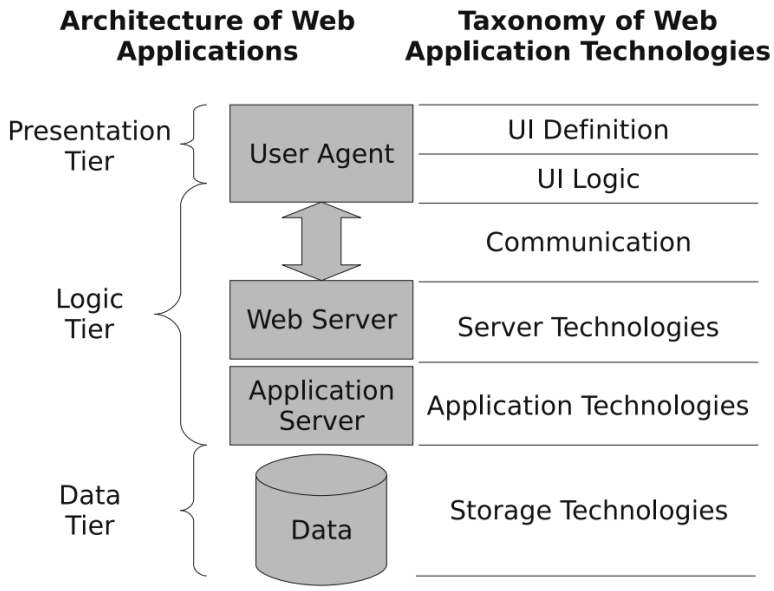

Figure 2.1 *Architecture of web applications and taxonomy of web application technologies. [44]*

The current trend amongst web applications seems to be shifting parts of the logic from the server into the client, with the wider use of RESTful APIs and the creation of thicker clients and thinner server. This trend has been partly rationalized by improving user experience. With thicker clients (enhanced with new technologies) it is possible to create richer interactions, and applications that do not require as many server round-trips as traditional web applications would. This shift and emerging standards also make it possible to use the application, even when there's no network connection, once the application has been cached, and presuming the application does not require network connection for its operation. The development of software becomes easier when using single paradigm, rather than dividing it into multiple languages and sites [44]. Probably the most widely known application for the end-user that represents the shift into thick client and thin server architecture, is Google's GMail.

This trend can partly be confirmed by looking at the current popularity of JavaScript. Figure 2.2 presents JavaScript being the most popular programming language over the fourth quarter of 2014, based on StackOverflow[1] conversations and GitHub[2] repositories [40].

The architecture of web application is undergoing changes and is transforming away

[1] `stackoverflow.com`
[2] `github.com`

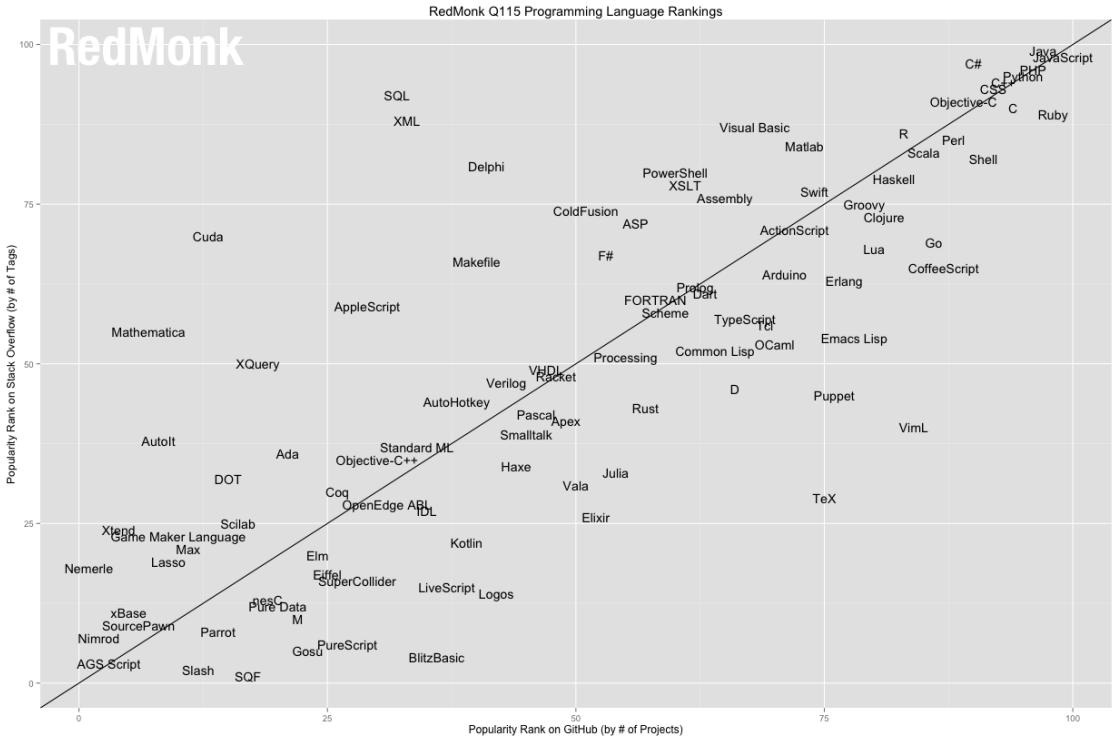

Figure 2.2 *Popularity of programming languages based on Stakcoverflow-tags and GitHub-repositories [40]*

from more traditional web applications. As parts of computing and also data-storage are shifting from the back end into the client-side, more and more engineering-work, or at least the established principles, need to shift from the back end to the client-side. Computer engineering principles and design patterns (proven and tested concepts) developed during the last thirty years should be taken advantage of when creating web applications, in order to create applications with high quality in a shorter timeframe.

2.2 Web Browser Domain

The main responsibility of the Web browser has been to present content the user has requested. Briefly explained, the way the browser achieves this is that the browser sends an HTTP GET-request to the web server (defined by the URL) after the server receives the request, it processes it and returns the appropriate response (HTTP Response, with appropriate status code and body). In turn, the browser receives the HTTP Response, processes it, and hopefully receives an HTML document in the response's body. After receiving the document, the browser parses it, fetches possible additional resources (for example images and stylesheets) and displays the

content in the browser's window. The resource is typically an HTML document, but it may also be a PDF file, image, or other type of content. We will look at the browser's actions in more detail in section 2.4.2.

To get a better understanding on how browsers have evolved, we will take a brief look into the web's origins. Tim Berners-Lee developed the first web browser in 1991, called *WorldWideWeb*; the browser was text-only, and served also as an HTML editor [2]. Soon after that came another text-only browser called *Lynx* [64]. In 1993 *Mosaic* was released, Mosaic introduced features that are still used in modern browsers, such as icons and bookmarks [38]. A company called Spyglass was created to commercialize Mosaic's technologies. Mosaic's author left to co-found their own company, which created the commercially succesful browser: *Netscape Navigator* [65]. In 1994 Tim Berners-Lee founded *W3C* to steer the development of Web standards and promote interoperability among web technologies. In 1995 Microsoft released *Internet Explorer* [66], which was based on code licenced from Spyglass. This started the era of the so-called Browser Wars, a period during which browser vendors competed over the domination of usage share in web browsers. During the mid 1990's a new closed source browser called *Opera* was introduced [67]. In 1998 Netscape open sourced their browser under the name *Mozilla* [68]. A few years later an open source browser called *Konqueror* was introduced [23], on which Apple's web browser *Safari* is based on [69]. Apple open sourced *WebKit*, which is a rendering engine used in Safari [90]. In 2008 Google announced the open source browser *Chromium* and *Chrome*, which is largely based on Chromium, adding Google's proprietary features to it, thus making it closed source [70]. Chromium and Chrome both have used WebKit as their rendering engine, but in 2014 they forked WebKit to create their own rendering engine *Blink*. In 2014 also Opera switched to use Blink [43]. Most of these browsers are illustrated in the timeline Figure 2.3 below.

During the last years, the use of mobile devices to browse the web has risen remarkably. Today more people have access to the Web by a mobile device than from a desktop computer connected to the Internet [55]. Currently there are five widely used browsers on desktop computers: Firefox, Internet Explorer, Chrome, Safari and Opera. When it comes to mobile devices, the main browsers are the Android Browser, iPhone (Safari), Opera Mini, Opera Mobile, UC Browser, Nokia S40/S60 Browsers and Chrome. Figure 2.4 presents the usage of different desktop browsers over time. [16]

For web browsers to work similarly, they follow the standards set by *World Wide Web Consortium (W3C)*. The standardization process happens somewhat simultaneously with the browser's implementations, which poses some challenges to the

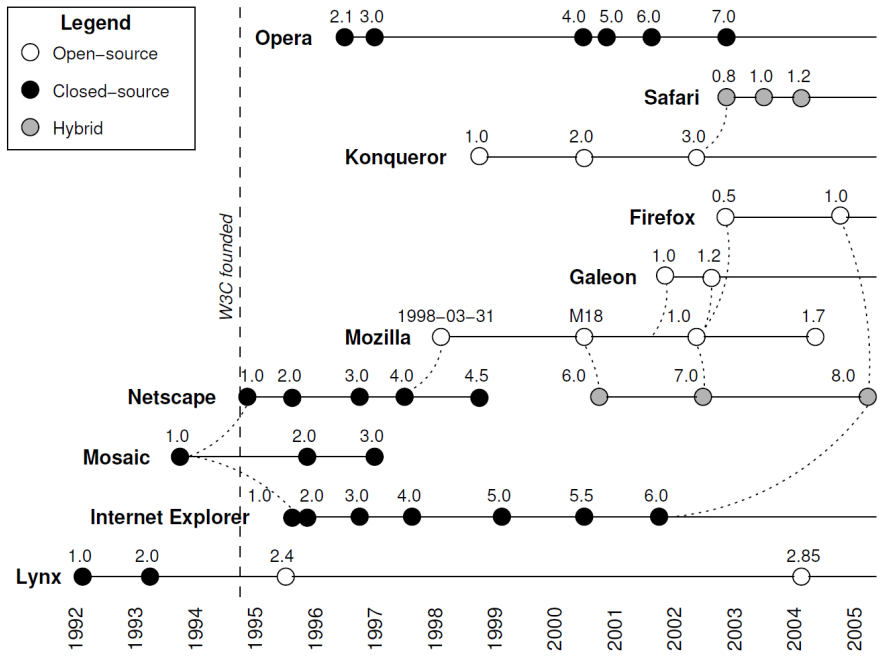

Figure 2.3 *Timeline picture of browsers evolution. [15]*

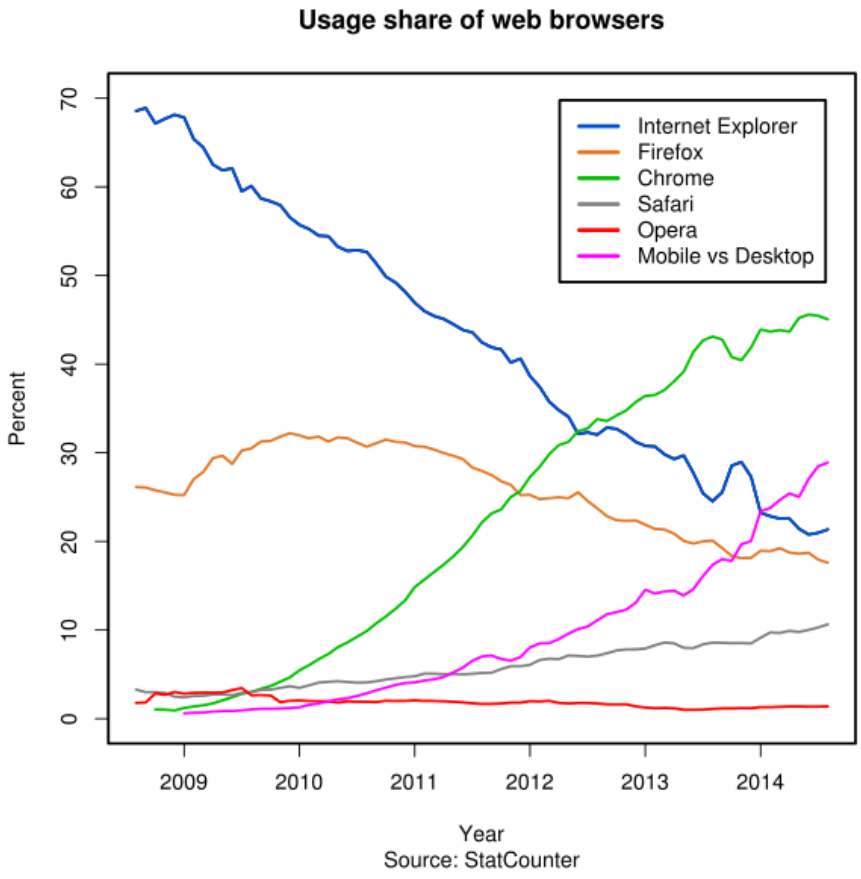

Figure 2.4 *Usage share of web browsers.[53]*

process. The standardization bodies do not want the implementations to happen before the specification is finished, in order to avoid a situation where web developers start relying on implementation details too early. Then again, W3C appreciates the feedback from real users and does not want to make complete specifications before the implementations and the author feedback. [56]

2.3 Technologies

In the previous sections we have described what browsers are made for and how they have evolved. In this chapter we will describe the standard technologies that are used when creating the client-side of the web applications. Since this thesis focuses on web applications that have thick clients and using the browser as the platform, we will not put large emphasis on the server-side of the application.

When developing client-side applications, there are mainly three different programming languages to be aware of. These languages are *HTML*, *CSS*, and *JavaScript*, which are designed in such a way that they have different responsibilities and goals, and are a way of enforcing the separation of concerns -concept. HTML describes the structure, CSS describes the presentation, and JavaScript the behaviour, as presented in Figure 2.5. In the following subsections we will go through them separately and how they are connected through Document Object Model (DOM), in addition to these we also take a brief look into HTML5.

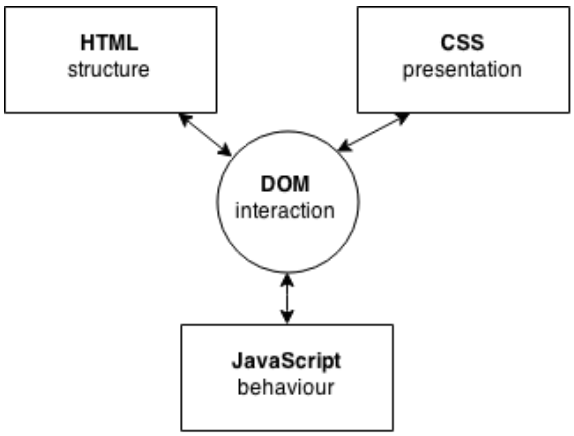

Figure 2.5 *Programming languages and their responsibilities in web applications*

2.3.1 HyperText Markup Language (HTML)

HTML is the standard markup language used in the World Wide Web. It was originally designed as a language for semantically describing scientific documents. Howe-

ver the design was very general, thus enabling it to be adopted over the last decades to describe a wide variety of documents and applications. [83]

HTML is a declarative language, used to create structured documents. These documents are formed by HTML elements. Syntactically HTML elements consist of tags enclosed in angle brackets. HTML elements are either paired, they have starting and closing tags; for example `<h1>Header</h1>`, or empty `<img>`. HTML-tags can have attributes, which are usually attribute-value pairs separated by the equality sign (=), and written in the start tag of the element, after the element's name, eg. `<img src=‘‘/images/example.png’’>`. In the following example we see an example of a simple HTML5 document, containing a separate stylesheet and a script file.

```
<!doctype html>
<html>
  <head>
      <title>Example</title>
      <link rel="stylesheet" href="presentation.css">
  </head>
  <body>
      <h1>Example</h1>
      <p>This is a paragraph</p>
      <p>Another paragraph.</p>
  </body>
  <script src="behaviour.js"></script>
</html>
```

2.3.2 Cascading Style Sheet (CSS)

Cascading Style Sheets (CSS) are used to define the presentation of the document written in a markup language (typically HTML). CSS is used to separate content from its represantation, a common design principle, and a specific instance of separation of concerns. CSS, like HTML is also a declarative language. The separation reduces complexity and repetition, improves content accessibility (by promoting semantically structured documents), provides flexibility and control.

CSS can be written into HTML document inline, or it can be written into a separate CSS file and referenced from a HTML-document. A Typical CSS file consists of a list of rules, which each contain a selector and a declaration block. The declaration block contains a collection of properties. Property is defined by specifying the property's name, followed by a semicolon and its value. The selector defines on which HTML elements these properties should be applied to.

CSS syntax is quite simple, using multiple English keywords to specify style properties. Here is a simple example of a CSS file, used to to define an outlook of the previously presented HTML-document.

```
body {
  margin-left: 15%;
  font-family: "sans-serif";
  font-size: 16px;
  color: rgb(40, 40, 40);
  background-color: rgb(250, 250, 250);
}

p {
  border: 1px solid rgb(40, 40, 40);
}
```

2.3.3 JavaScript

JavaScript programming language plays a large role in today's web applications. Its importance has been growing in the past years, as the browser has been transforming from a document viewer into a runtime environment. Client-side web applications' functionality is implemented with JavaScript. In order to analyse web application's memory consumption in the following chapters, we will take a brief look into the JavaScript programming language. For more in depth references on JavaScript, see the following books: D. Flanagan's *JavaScript The Definitive Guide* [57], and D. Crockford's *JavaScript The Good Parts* [8].

JavaScript is a dynamic programming language, most widely known for its use in client-side scripting. It was developed at Netscape by Brendan Eich, at the same time as non standardized DOM (DOM Level 0, or Legacy DOM) was being developed. JavaScript was originally released in Netscape Navigator's version 2.0B in 1995 to support simple client-side scripting. Later during the *Browser Wars*, Microsoft created their own scripting language called JScript. JavaScript and JScript are implementations of ECMAScript, which is the standardized version by Ecma International. A standardized version was defined by Ecma International, in order for the different implementations of the standard to behave similarly. ECMAScript was created to capture the common elements of JavaScript and JScript [42]. There have been five editions of ECMAScript (allthough the 4th edition was abandoned), the current edition is 5.1, and the next edition will be ECMASript edition 6 which is already feature ready.

JavaScript has had a reputation of being a "toy" language, the reasons largely stemming from its use and beginnings in simple scripting [47]. JavaScript supports multiple programming-paradigms: object-oriented, imperative, and functional programming styles. JavaScript's use is not limited to client-side scripting, it is a general-purpore programming language and lately the use of JavaScript has spread to other areas. JavaScript's usage growth has been especially significant on the server-side. Currently a very popular runtime environment for server-side JavaScript applications is *node.js* [60]. During the work on this thesis we implemented our demo application's server-side with *node.js* as well.

JavaScript is known for its permissive nature: this design decision was made in order to make the language easier for beginners. But actually in doing so, it makes it extremely difficult in finding certain types of errors. One example of the permissiveness is that the developer does not have to declare variables explicitly. If the `var` keyword is omitted, the variable will automatically be published to the *global-scope*. This also means that if a developer mistypes a variable's name while placing a value to it, a new variable will be created into global scope (implicit global). The more traditional way would be to throw an Error, which would make it easier for the developer to find the root cause for other errors. Afterwards implied globals have been criticized, for example by D. Crockford [8], and in terms of memory management they are also very harmful. Fortunately later on this permissiveness has been limited. Today these kinds of errors can be caught more easily with *strict mode*, introduced in ECMAScript edition 5 [30]. The permissiveness of the language, its name and its superficial syntactic resemblance to Java have contributed to the notion that developers do not have to study JavaScript before its use. D. Crockford has written that JavaScript is the "The World's Most Misunderstood Programming Language" [5], since JavaScript's importance is growing and it has its share of characteristics, it is important to really understand the language and its features.

JavaScript's data types can be divided into primitive and non-primitive types. Primitive types are data types that are not Objects. There are six primitive data types in JavaScript: `String`, `Number`, `Boolean`, `Null`, `undefined` and `Symbol` (which has been introduced in ECMAScript 2015). `Strings`, `Numbers`, `Booleans` and `Symbols` are object-like, since they have methods, but in comparison to non-primitive types the main difference is that they are immutable. Objects in JavaScript are mutable keyed collections. In JavaScript functions, arrays, regular expressions and of course objects are objects.

JavaScript was designed to be a prototype-based scripting language. Therefore it did not include the notion of *Class*, until the latest edition of the ECMA-262 standard,

which we will have a closer look at in subsection 3.4.1. Before the introduction of classes into JavaScript, inheritance was done by prototypal inheritance. Every object is linked to a prototype object, from which it can inherit properties. If an object is created from a so called object literal, it will be linked to `Object.prototype`, which is an Object that comes standard with JavaScript.

Arrays are actually Objects that have numerical keys, arrays are inherited from `Array.prototype`. The main difference with arrays and standard objects is that arrays have a special connection with the integer keyed properties and the property `length`. [31]

A very important design decision in JavaScript is that it has first-class functions. Functions in JavaScript are actually callable objects and they can be manipulated and passed around just like any other object. To be more specific Functions are Function objects inherited from `Function.prototype`, which in turn is inherited from `Object.prototype`. [8]

JavaScript has a function scope, which means that paramateres and variables defined in a function are not visible outside of the function. A variable defined anywhere in the function is visibile everywhere in the function. However ECMAScript edition 6 will introduce block scoped variables to JavaScript. The new edition of the standard will add the `let`-keyword which makes it possible to declare block scoped variables. Because of function scope D. Crockford has advised that variables and parameters that will be used within a function are declared at the top of the function body [8]. This improves code's readability by making the scope of the variable clear, and also avoids confusion caused by feature in JavaScript called *variable hoisting* [32].

JavaScript makes it possible for functions to have nested functions. The function defined within a function has not only access to its own variables and parameters, but to the variables and parameters of the functions it is nested within. Function object created by a function literal, contains a link to its outer context. These inner functions are usually called closures, meaning that the inner function closes the parents' lexical environment, resulting in a closed expression. In the following example a simple closure is presented:

```
function foo() {
  var lrgString = new Array(10000000).join("z");
  var baz = function () {
    // here we have access to parent functions variables
    return "result is: " + lrgString;
  };
  return baz;
```

```
}

var bar = foo();
// bar contains a reference to baz
var resultString = bar();
// resultString equals "result is: zzzzzzzzz...."
```

Because JavaScript has first-class functions, it is possible for the closure to have longer lifespan than its parent. In this thesis our focus is on JavaScript applications' memory management and it should be noted that closures are an easy way of obfuscating memory references. Since closures can obfuscate the references an object maintains, references to unneeded JavaScript Objects or DOM elements may be maintained unnecessarily.

2.3.4 Document Object Model (DOM)

Document Object Model is a platform- and language-independent convention for representing and interacting with the content, structure and style of documents. The DOM provides a programming API for documents. It provides a structured presentation of the document it models. DOM is used to describe structured documents, like HTML, XML, and SVG. These structured document's DOM nodes often create a tree-like structure, which is sometimes called a *DOM tree*. In the following Figure 2.6 is a presentation of the DOM tree, based on the document presented in subsection 2.3.1.

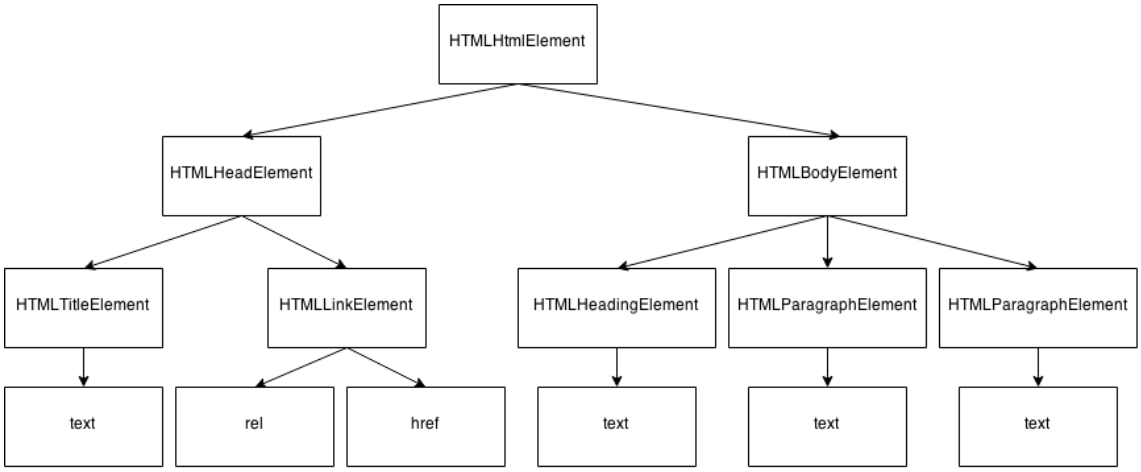

Figure 2.6 *DOM representation of a previously presented HTML-document*

The objects in the tree can be manipulated and addressed by using the methods

of the objects. DOM provides an Application Programming Interface (API) for interacting with the document and its objects, thus enabling the modification of the document with client-side scripting [88]. The document contains JavaScript, which can perform complex modifications to it, through the Document Object Model. This in turn means that the browser's represantation of a document in memory is a cross-language data structure between the low level native code (browser), and garbage collected JavaScript [36].

2.3.5 HTML5 and friends

HTML5 is the final and complete fifth revision of the HTML standard of the W3C. The standard includes, among other things, video-, audio- and canvas elements, which enable richer multimedia experiences on the web. In addition to this standard, HTML5 is also widely used as an umberella term (sometimes referred to as HTML5 and friends), including several technologies that allow more diverse and powerful web sites and applications [33]. These other technologies often refer to technologies such as CSS3, SVG, Web Workers, WebGL, Offline support, WebRTC and Device API's (Geolocation, Orientation, User Media). These new technologies provide the basic building blocks for building desktop-like applications on the browser.

2.4 Web-browser as an application platform

2.4.1 Motivation

The motivation behind the shift from conventional binary programs into web applications has been discussed with greater detail in Mikkonen and Taivalsaari's articles *The Death To Binary Software* [47] and *Reports Of Web's Death Are Greatly Exaggerated* [50]. In this section, the main motivators for this paradigm shift into web based software are described briefly. In the article *Reports Of Web's Death Are Greatly Exaggerated*, Taivalsaari and Mikkonen provide five motivators for the shift from traditional binary programs into web applications.

1. No manual installation or manual upgrades.
2. Instant worldwide deployment.
3. Open application formats.
4. Platform independence.

5. Ubiquitous seamless access to data.

These items may seem like quite obvious benefits that the web has to offer for the end-user, yet each of them offer remarkable possibilities. For example, the first item makes it possible to do continuous integration and A/B testing, which makes it possible to improve the product continuosly and to deploy newer versions with larger frequency. It becomes extremely difficult for conventional binary programs to compete with web based applications, on which software distribution is effectively free, and the effort of taking a new software system into use is almost nonexistent [47].

There are also other, somewhat more ideological motivators for the creation of a universal application platform for the end-user. These motivators are however also linked to the items listed above, including Open Application formats, Platform independce and Instant worldwide deployment. These motivators should also be taken into account, since the Web is not only a technological invention, but also a cultural phenomenon [21]. The Web has been built around egalitarian and decentralized principles. These principles have been discussed with greater detail in *Mozilla Manifesto* [37] and Tim Berners-Lee's article *Long Live The Web* [1]. The use of Open Standards as the basis of an application platform promotes a more scientific approach for the evolution of these standards, whereby new work can be based on previous works. This standardization process is open to participants and inspectors. The development of standards becomes an iterative process, which improves gradually. Gradual improvement is not guaranteed to happen if the development is solely dependent on a proprietary vendor [1]. Since iterative processes have been a proven, largely used concept when creating software, they could be useful when creating standards as well. The use of open standards and various stakeholders makes the implementation more future-proof compared to native applications. Therefore with the use of Open Standards it is easier to avoid vendor-locks, which in turn benefits the consumers and end-users.

The rise in the popularity of the *Bring Your Own Device* policy [73] can also be seen to further promote the idea of the universal application platform. The same application could be used on different platforms, even on currently non existent ones, without having them to be explicitly built for the system at hand.

Another motivator is also the large number of platforms on which provokingly the only shared feature is that most of these include, at least somewhat, a standard compliant web browser. The ideal situation for an application developer, when creating applications, would be to create the application once and deploy it on every

platform. Currently developers have to write source code for at least most of the platforms they are targeting, be it Desktop (*Windows, OS X, Linux*), or Mobile (*iOS, Android, Windows Phone, Blackberry, Symbian, Meego, Baida, WebOS, Sailfish, Tizen*, etc.). There are solutions like *Phonegap*[3] and *Adobe Air*[4] that make it possible to write the software once and run it on the supported platforms. Phonegap relies on web technologies and therefore it does not provide a real solution to the problem. Adobe Air, on the other hand, relies on Adobe's proprietary technologies including ActionScript programming language (a dialect of ECMAScript). Currently there are multiple platforms that rely on web technologies, for example *ChromeOS*[5], *Tizen*[6] and *WebOS*[7].

The motivation for using the browser as a platform can also be seen by the use of browser plugins from the late 1990's onwards. Several different technologies, including *JavaFX* [8], *Adobe Flash*[9] and *Microsoft Silverlight* [10], have been used to develop so called Rich Internet Applications (RIAs) [74]. These technologies share some common characteristics, like addressing the issue that Web technologies were not, at least at the time, well suited for rich application development. Another important characteristic of these technologies was that their use required separate proprietary browser Plug-In installation, which possibly contributed to their downfall.

2.4.2 Browsers' working principles

To understand web applications' memory consumption more thoroughly, we will take a brief look into browser operations and how they manage their memory allocations. T. Garsiel has written a book about Browsers' operations, called *How Browsers Work* [16], in which she describes a browsers operations with greater detail. This subsection is largely based on that book.

Information about browsers' operations is important for web developers in order for them to make better decisions and know the justifications behind the development best practices. We will now look at the browsers architecture, with a focus mostly on performance and memory management in mind.

[3] http://phonegap.com/
[4] https://get.adobe.com/air/
[5] http://en.wikipedia.org/wiki/Chrome_OS
[6] https://www.tizen.org/
[7] https://www.openwebosproject.org/
[8] http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html
[9] https://www.adobe.com/products/flashplayer.html
[10] http://www.microsoft.com/silverlight/

Even though Web browsers are developed by different companies with different objectives and goals, they share a number of features since they are designed to support the same standards. The features of the browsers are not standardized, but instead their features have been developed convergently. The interoperability between browsers has improved greatly in comparison to the situation a decade ago.

A. Grosskurth and M. W. Godfrey have studied the implementations of two well known open source browsers (*Mozilla* and *Konqueror*). They derived a reference architecture based on these browsers implementations and documentations. They then validated the reference architecture against two other browser implementations (*Safari*, *Lynx*). During the research, they found that the browsers' operations are divided between different components. The high level components are presented in the Figure 2.7. [15]

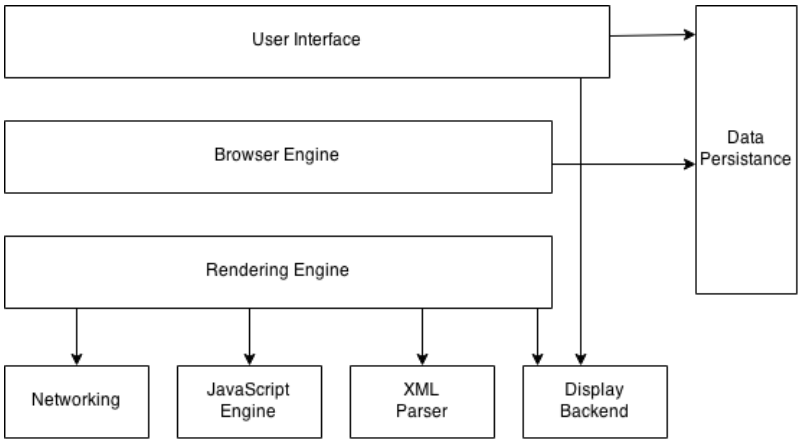

Figure 2.7 *Web Browsers High Level Architecture. [15]*

Browsers have similarities in their application flows. Here we take a brief look at how the browser turns the resources into interactive application. The browser's rendering engine will first start parsing the HTML document. It turns the HTML tags into DOM nodes and into a DOM tree, sometimes called a *content tree*. Then the rendering engine parses the style data from CSS files and style elements. The styling information is turned into another tree called *render tree*. The Render tree contains rectangles with visual attributes like dimensions and colour, in the correct order to be displayed in the screen. Next a *layout*-process is done (sometimes called *reflow*), during which each node is given the exact coordinates on where it should be displayed on the screen. After layout *painting* is done, here render tree is traversed and each node will be painted using the UI back end layer. This flow is presented in Figure 2.8.

It should be mentioned that this is a gradual process in order to improve user

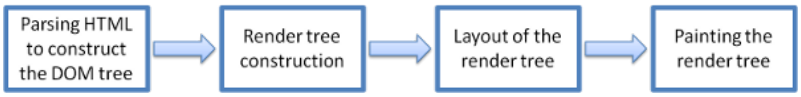

Figure 2.8 *Rendering engine basic flow [16]*

experience. The building and layout of the rendering tree happens simultaneously as the HTML document is being parsed. To have a better idea on how browsers parse the documents and/or stylesheets, and then construct the trees, see T. Garsiel's *How Browsers Work* [16].

The rendering engine gives the scripts for the JavaScript engine to process in the same order and at the exact moment as *<script>* tags appear in the document. The rendering engine stops the parsing of the document in order to execute the script. If the script was external it has to be fetched from the network. This is also done synchronously, which explains why it is often proposed to put the scripts at the bottom of the document. This was the way script processing was done for many years and it is also specified in HTML 4 and HTML 5. It is possible to mark the script as *defer*, which does not halt the document parsing and will execute the script after the document has been parsed. HTML5 also makes it possible to mark the script as asynchronous so it will be parsed and executed by a different thread. [16]

The DOM tree is effectively a cross-language object, shared between the browser (its native code) and JavaScript the document contains. It is important that the browser does not remove DOM objects while they are still reachable from either JavaScript or native code. This issue would cause *use-after-free* bugs, which often produce exploitable security holes. In order for the browsers to solve such issues, most existing browsers use reference counting (subsection 3.2.1) to track the references to underlying low-level DOM objects. For example, when a JavaScript-code fetches a DOM object via DOMs `getElementById()`, the browser creates a "reflector" object into the JavaScript engine's heap (see Figure 2.9). After the JavaScript engine's garbage collector determines that this reflector has become garbage, it will be destroyed and the reference count of the DOM object will be decreased. Once the reference count reaches zero, the DOM object will be removed. [36]

This cross-language reflector scheme solves the *use-after-free* bugs, but at the same time introduces a new problem, which is caused by cyclic references. In order for the browsers' memory footprint to stay as small as possible, browsers' need to destroy the objects as soon as they are no longer needed. When using naive reference counting, it is trivial to create cross-language cyclic references from JavaScript. This kind of

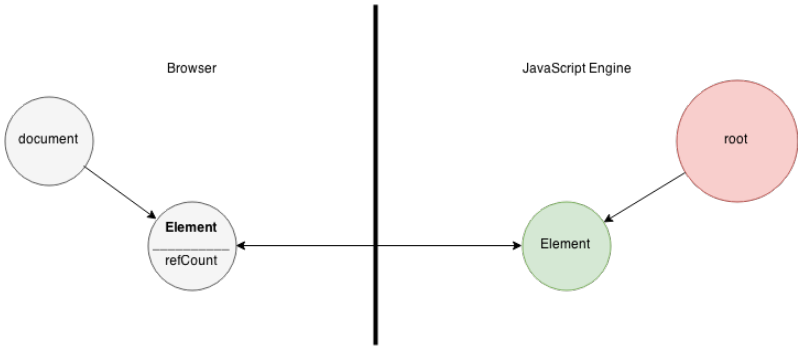

Figure 2.9 *Reflector scheme*

cyclic reference is being created when a DOM element has an eventListener, whose execution adds a property to the event, which in turn references into the element. This is illustrated in Figure 2.10 of the memory graph, which is based on the following code example: [36]

```
element.addEventListener('load', function (event) {
  event.originalTarget = element;
}, false);
```

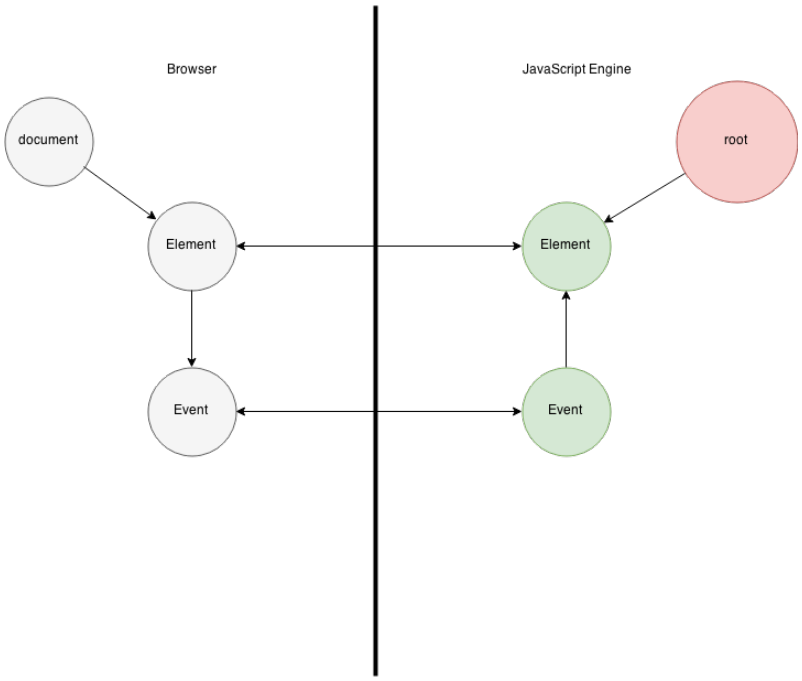

Figure 2.10 *DOM Cyclic Reference example*

These kinds of cyclic references cannot be cleared by naive reference counters and JavaScript's garbage collection does not trace through browsers' pointers, so these objects cannot be freed. Existing browsers have developed different ways to solve

these issues. Some leak the memory, some try to manually break possible cycles and some implement a cycle collection algorithm on top of reference counting. [36]

2.4.3 JavaScript Engine

Every current browser includes a piece of software called JavaScript engine. Browsers' JavaScript engine's implementations have shifted from interpretation into *Just In Time (JIT) compilation*, which has largely contributed to the JavaScript engines' improved performance over the past years [58]. In this section we will look at JavaScript engines only briefly to build a general understanding of how they manage JavaScript's memory allocations and deallocations. Different browsers have their own implementations of JavaScript engines. Table 2.1 provides the names of JavaScript engines used by the most popular browsers.

Table 2.1 *JavaScript Engines used by the most popular browsers*

JavaScript engine	**Browsers**
Spidermonkey	Mozilla Firefox
v8	Google Chrome, Chromium, Opera
JavaScriptCore	Safari
Chakra	Internet Explorer

A reference architecture study for JavaScript engines' implementations has not yet been carried out, this would make it possible to refer to JavaScript engines in general without going into implementation details. Therefore we will briefly discuss a specific JavaScript engine implementation, *v8*. This was chosen, since it is widely used and during this thesis we relied on a Chrome's proprietary API (*performance.memory*), which exposes v8's heap usage data for the application developer.

As we described in subsection 2.3.3, JavaScript's memory is automatically managed. This means that JavaScript engine implementation has to manage the memory allocations and deallocations for the application. v8 allocates memory at a runtime from the *heap* for JavaScript objects [18]. In v8, objects that require a large amount of memory, like *arrays* and *strings*, only store a wrapper to the data in the heap, and store their data in the renderer's memory [19]. Garbage collection takes care of deallocating the memory of the objects that will no longer be needed.

2.4.4 Single Page Applications

Over the past decades, interactions in the Web have revolved around forms and documents, which have had many limitations, the most important being a lack of

rich user interactions and excessive network round-trips. Transitioning into the so-called Web 2.0 started a shift into more dynamic documents, where *Asynchronous JavaScript and XML* (*AJAX*) was being used for fetching resources from the server, without loading the whole document. Single Page Applications (*SPA*) can be seen as expanding the same idea even further. In this section we will look at the Single Page Applications, their characteristics, and a few Single Page Application Frameworks. We will then describe how these characteristics, and JavaScript's memory management are related.

In recent years Single Page-Applications have been developed in order to develop more native-like applications that can be used in the web browser. Single Page Applications address the arcane navigation model of the Web, which has developed from the Web's document oriented origins. This is done in order to create richer interactions and improve performance, thus providing better user experience. The transformation from interlinked documents into more interactive web applications has changed the architecture of web applications greatly. Navigation logic is implemented with JavaScript, so that when the user navigates around the application, browser does not fetch complete HTML documents from the server. Instead the resources, for example needed to represent the user interface (*UI*), are already loaded to the browser, and only the dynamic content has to be fetched. [46]

Single Page Applications share some common characteristics. They improve the overall performance by loading most of the applications' static resources (*images*, *CSS*, and *JavaScript* files) during the initial load, to reduce the number of server round trips. When most of the resources are loaded during the initialization phase, the communication that happens with the server is mostly fetching dynamic content by AJAX-requests and/or WebSockets.

Client-side rendering enables richer interactions compared to server-side rendering. Server-side rendering will quickly become cumbersome when there are multiple components on a page, and those components can have multiple intermediate states (eg. menu open, menu clicked, menu item selected, menu item clicked). These small view states cannot be mapped very well to URLs. Also retrieving all of these intermediate views from the server would add unnecessary server round-trips, when compared to client-side rendering. [46]

Single page application frameworks have been developed for a number of reasons, but one of their shared reasons seems to be enforcing engineering principles, one of these being separation of concerns. In the fast moving world of JavaScript frameworks and libraries, the amount of frameworks can be overwhelming. Few mature

frameworks for Single Page Application development include Google's *Angular.js*[11] and DocumentCloud's *Backbone.js*[12].

Memory management's role increases when developing Single Page Applications, compared to more traditional, document-based web applications. This is because browser's JavaScript engine's global variables are flushed on page load and refresh events, which by design happens less frequently in Single Page Applications, when compared to more traditional web applications.

[11] Angular.js `https://angularjs.org/`
[12] Backbone.js `http://backbonejs.org/`

3. WEB APPLICATION'S MEMORY CONSUMPTION

3.1 Dynamically Managed Memory

As we have explained in the previous chapter, modern web applications behaviour is implemented with JavaScript. Like many modern dynamic programming languages, JavaScript's memory is *automatically* managed, therefore application developers do not have to explicitly deallocate memory. Dynamically managed memory avoids multiple memory-related problems, including, double frees, premature frees, memory safety issues, and certain types of memory leaks. The rationale behind dynamically managed memory is that, it frees the developer from investigating aforementioned problems, increasing developers productivity [51]. In JavaScript's case developers do not have *any* control over how memory is managed, since the ECMAScript specification does not define any interface to the implementation's garbage collector [58].

Dynamically managed memory is an abstraction, and as with all abstractions, sometimes the implementation leaks through the abstraction. This leaking reveals implementation details the abstraction is not able to hide. J. Spolsky calls this the law of leaky abstractions, *"All non-trivial abstractions, to some degree are leaky."*. Because of leaky abstractions, it is important to understand what is actually abstracted by garbage collection and what to do when the abstraction leaks. [45]

Dynamically managed memory makes the language easier for the developer, but it also obfuscates the developers need to think about memory; its management and therefore consumption. Garbage Collection is by no means a substitute for effective memory management at application level. Applications developed with Garbage Collected languages, suffer from some of the same problems that applications developed with manually managed memory do, such as memory leaks. In addition to this, Garbage Collection also pauses the application from time to time, in order to collect the garbage. These are the moments when the application developer experiences the leaky abstraction. The more memory an application consumes, the more disruptive the Garbage Collection stops become. Thus creating longer and

more frequent pauses during the applications runtime. JavaScript engine stops the application when doing garbage collection. This is unfavorable for application's performance, and it can also cause unresponsiveness in the application's user interface. Other symtomps in unsuccesful memory management include, failure, and holding references to limited resources. [54]

During this thesis we are especially interested in avoiding failures caused by unsuccessful memory management, and creating responsive user interfaces. These are the interests because we are looking into developing robust, mission critical, long-running applications with web technologies. These requirements map to memory management's requirements as presented in Table 3.1:

Table 3.1 *Requirements For Memory Management*

Application Requirement	**Application's Memory Management**
Robustness, Long-Running	Application does not leak memory
Mission Criticality, Responsiveness	Garbage Collection is not intrusive

From application developer's point of view, the intrusiviness of the garbage collector, means that the application should not allocate excessive amounts of memory continuously. These requirements are related to the objects allocated by the application, and their lilfespans.

In the next section, we will address garbage collectors working principles, and implementations. Allthough in general, developers should not have to bother considering the virtual machine's implementation details, when building or designing applications on top of it, memory-related aspects cannot be ignored in mission critical; robust, long-running, applications.

3.2 Garbage Collection

To effectively manage application's memory, developers should understand how JavaScript engine manages memory allocations and deallocations. High level programming languages' runtime environment contains a piece of software called garbage collector, and its job is to track memory allocation and use in order to find when allocated memory is no longer used. Then the garbage collector will make the memory available for future allocations, or release the memory back to the operating system. The genaral problem of knowing when a certain piece of memory is no longer needed is *undecidable* (similar to the halting problem). Therefore the implementations of Garbage Collectors make approximations when trying to solve the problem. [34]

Garbage collection algorithms usually rely on the notion of references when deciding when memory is unused. Usefull way to conceptualize memory management in JavaScript from developer's point of view is to think of the memory as a graph, as presented in Figure 3.1. Primitive types (Numbers, Boolean, Strings, Symbol) in JavaScript are allways the terminating nodes (or leafs) in the graph. In JavaScript references to primitive types are allways stored in objects, the references are represented by arrows in the following figure. The next subsections will explain briefly two different Garbage Collection Algorithms and their limitations.

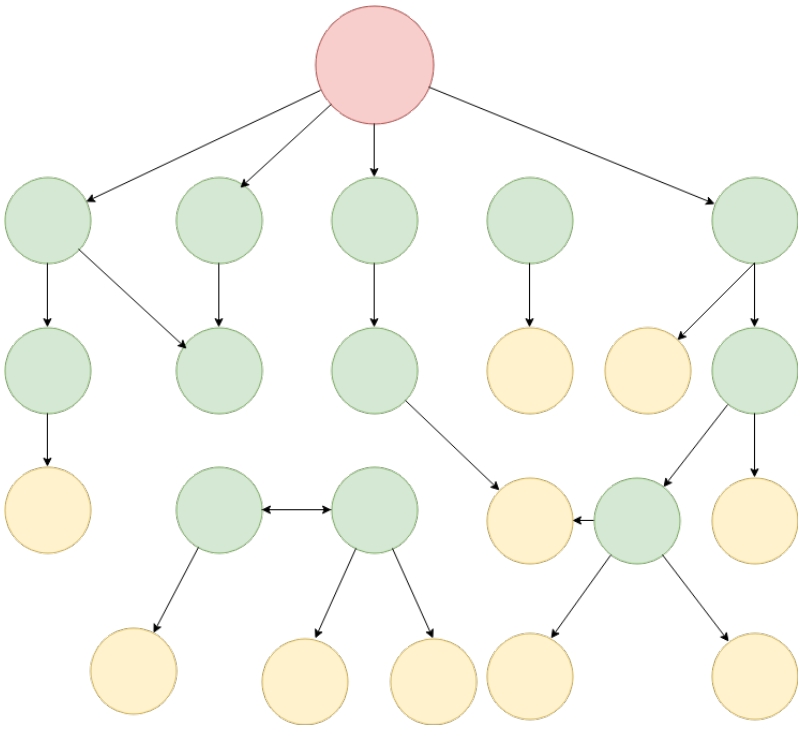

Figure 3.1 *Memory Graph*

It should be noted that the typical implementation of JavaScript engines garbage collector, is such that garbage collection is instantiated by memory allocation, not by dereferencing objects. So every time an application allocates new memory, the closer the next garbage collection becomes.

3.2.1 Reference counting garbage collector

The most naive algorithm in deciding whether a certain object (or value) is no longer used is Reference Counting Garbage Collector. This reduces the problem of *whether an object is no longer used*, into *whether an object has references to it.* Naive Reference Counting Garbage Collectors are known to be implemented for DOM objects in IE6 and IE7. [34]

The implementations have limitations when it comes to cyclic references, which makes it quite simple to create memory leaks. In the following example we present a way of leaking memory in IE6 and IE7. This memory can only be reclaimed once the browser is restarted. The cyclic reference that is presented in the following example is the work of a closure, which has been created between the anonymous function (event handler callback, more precicely its lexical scope), and the DOM element. [28]

```
function addHandler() {
    var el = document.getElementById('el');
    el.addEventListener('click', function () {
        el.style.backgroundColor = 'red';
    }, false);
}
```

Even if the DOM element is removed from the DOM tree, IE6 or IE7 cannot deallocate it, since it still remains referenced from the function object's lexical scope. The previous example also demonstrates the complexities that are hidden between DOM's and JavaScript engines mutual memory management.

We can also take a look at the previously presented example's memory graph from reference counting garbage collector's point of view. In Figure 3.2, the garbage is outlined in red dotted line. There are detached nodes, but since both of them have references, even though from each other, they are not considered garbage. It should be noted that this is not quite accurate in the IE6 and IE7 memory leak situation, since IE's DOM elements are not managed by JScript engine's garbage collector, but instead they have their own memory manager [7].

Because of their simplicity reference counting garbage collectors are easy to implement and cause less overhead when compared to the alternative, tracing garbage collector. Naive reference counting garbage collectors are widely used for example in, *Objective-C*, *Perl*, *Delphi*, *PHP*, and *Swift* [77]. It should be noted that some of the previous (including Objective-C, Swift (which use ARC[1])) insert the memory deallocations during compilation, and therefore do not perform garbage collection at runtime per se.

[1]Automatic Reference Counting `http://en.wikipedia.org/wiki/Automatic_Reference_Counting`

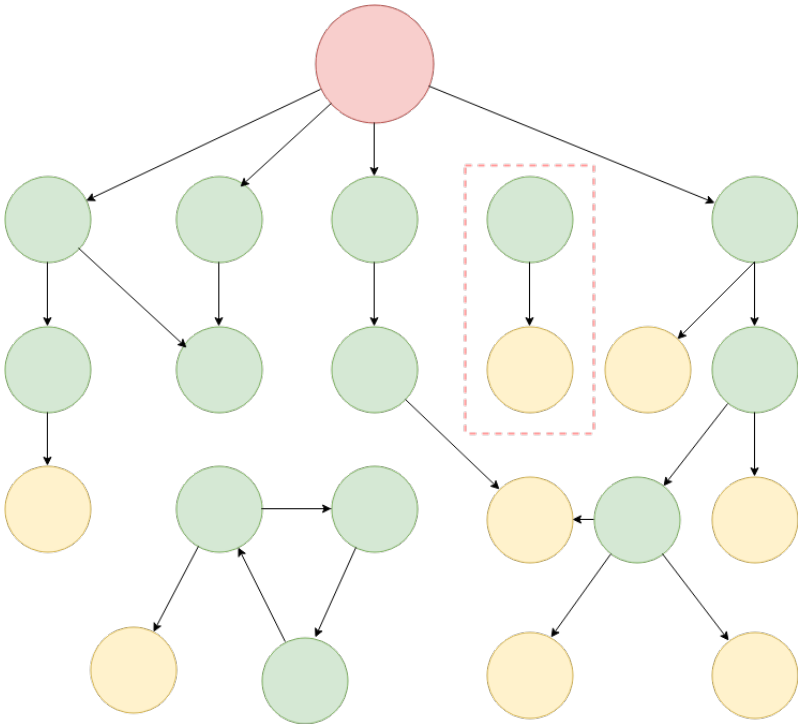

Figure 3.2 *Memory Graph Reference Counting Garbage Collector*

3.2.2 Tracing garbage collector

Tracing Garbage Collectors also uses references to detect when object is no longer needed, its rationale is as follows; if an object cannot be reached, it becomes garbage. Most implementations of Tracing Garbage Collectors use Mark And Sweep Algorithm for going through the references. Mark And Sweep Algorithm reduces the problem of *whether an object is no longer used* into *whether an object is reachable* (Figure 3.3). Mark And Sweep Algorithm assumes the knowledge of a root object. When we are addressing client-side JavaScript, the root object is the global object (DOM window [29]).

Mark And Sweep Collector has one limitation: objects should be explicitly declared unreachable, in order to make them eligble for garbage collection. This can be done by dereferencing variables, or by controlling variables lifetime by its scope.

Every browser released later than 2012, ships with Mark and Sweep garbage collector [34]. Therefore currently memory leaks definition in Web applications client-side can be reduced to, *holding references to unneeded objects*.

When addressing client-side web applications, the root node is the window-object. Therefore JavaScript's global variables are actually properties of the window object.

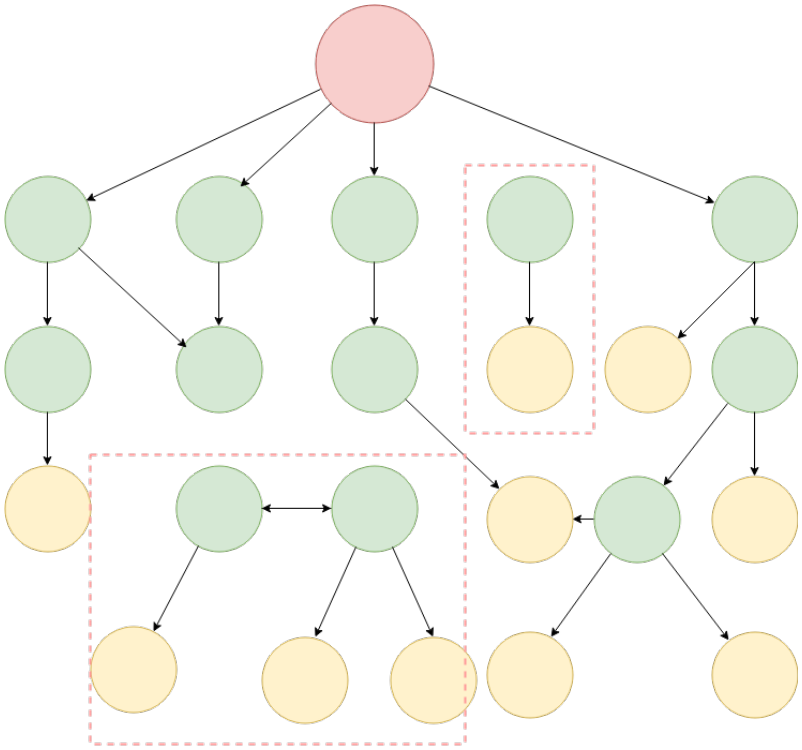

Figure 3.3 *Memory Graph Tracing Garbage Collector*

The root-node's memory management is out of application developers control, since it is handled by the browser. Root node is created when the page is loaded, and removed when the page is unloaded.

3.2.3 Generational

Some Garbage Collectors are known to divide values based on their age into two groups. These kind of Garbage Collectors are called *Generational Garbage Collectors*, and they are known to be implemented in *Spidermonkey* (Firefox) [35] and *v8* (Chrome, Opera) [18]. There is currently work under way in building *JavaScriptCore* (Safari, WebKit) a generational garbage collector [62]. The terms used to describe the groups vary between the implementations, but they are quite self explatory: *Young* (v8) / *Nursery* (Spidermonkey) and *Old* (v8) / *Tenured* (Spidermonkey). The division is done, to use the information of how long value has survived to deduce how long it will survive. This is empirical approach, which has been rationalized by analyzing many JavaScript applications and finding out that it is most likely for the recently allocated objects to become unreachable quickly (also known as *generational hypothesis*, or *infant mortality*). Values that have retained over few garbage collections, will be promoted to Old/Tenured generation.

The differences between the two generations are that garbage collection is faster and more frequent for the young generation. The collection of young generation is faster, because the cost of garbage collection is proportional to the number of live objects. This is lower in young generation, when generational hypothesis holds. Garbage Collector is able to free larger percent of memory from the young generation (death rate 80%). [41]

JavaScript engines also have to manage heap's fragmentation. This is done by periodically ordering Old generations objects. For this task v8 uses Mark-compact algorithm [54].

3.2.4 Incremental vs stop-the-world

Garbage collectors can be further categorized based on whether they divide the garbage collection cycle into discrete phases or not. A garbage collector that halts the execution of the program completely to run a garbage collection cycle, is called stop-the-world garbage collector. This method guarantees that new objects are not created during the garbage collection process. As the name suggests the program is not able to do anything during the garbage collection, this can cause problems with interactive programs, since unresponsiveness has a negative effect on user experience.

In comparison to stop-the-world garbage collectors, there are incremental, and concurrent garbage collectors that execute their garbage collection cycle in discrete phases. This makes the stops shorter, but more frequent, however the sum of the incremental phases takes longer than stop-the-world garabge collections pause. Concurrent garbage collectors may run simultaneously as the program is being executed. They only need to stop the execution for analyzing the program's heap.

3.3 Memory Leaks in Web Applications

We have now concluded that JavaScript's memory is automatically managed, therefore JavaScript applications do not have memory leaks in the same sense that C/C++ programs may have. In current JavaScript engines implementations where tracing garbage collectors are being used, JavaScript's memory leaks definition can be reduced into; *holding references to unneeded resources*. Apple's *Advanced Memory Management Programming Guide*, describes a memory leak as follows: *"A memory leak is where allocated memory is not freed, eventhough it is never used again. Leaks cause your application to use ever-increasing amounts of memory, which in turn may result in poor system performance or your application being terminated."*

[78] This guide is not about JavaScript per se, but however it accurately describes memory leaks in automatically managed languages. That being said, memory leaks in web applications are actually logical programming errors. Here is an example regarding how to leak memory in web application:

```
lightController.view = document.createElement('div');
// add the element to the DOM tree
lightControllerCollection.appendChild(lightController.view);

....

// remove child elements from the DOM tree
lightControllerCollection.removeAllChildren();
```

Even though we have removed all the children from the collection, we still have a live reference to the DOM node from the `lightController` object's `view` property, which prevents the DOM node from being removed, even though it has been detached from the DOM tree. Uncleared references are the reason that memory is being leaked in JavaScript. These references may be pointing to pure JavaScript objects / variables, or to DOM node's wrappers (which are also JavaScript objects), as in the previous example. Leaks that are caused by references to DOM nodes, which are not part of the DOM tree are called *DOM leaks*, during this thesis. If there is a live DOM node wrapper in JavaScript heap, the browser does not remove the corresponding DOM object from the browser's memory, (as described in section 2.4.2). If the DOM object has its own subtree, this also cannot be removed. DOM nodes can also have indirect references from other DOM nodes. Few examples of indirect references from the DOM node's properties, such as `nextSibling, previousSibling, parentNode, firstChild` [89]. For this reason it is advised to not hold refences to DOM nodes for extended periods.

As the browser is transforming into a runtime environment, it is becoming ever more important to have the required tools and standards in place, to enable the development of applications that manage their memory allocations succesfully. Also today applications made with web technologies are run on various platforms, including memory critical mobile devices. On these devices succesful memory management is even more important in order to improve performance, and to use the limited memory effectively. In this thesis we are interested in mission critical web applications, that are expected to be running for long periods of time, (where the use is measured in months rather than hours). In this use case the role of succesful memory management becomes compulsory.

Web application's memory leaks have been around from the the introduction of DOM and JavaScript. Previously there has not been large interest in these issues, for possibly the following of reasons:

- Web applications with thick clients have not been around for very long time.
- The lack of established *web engineering* dicipline.
- Traditional web applications are not usually mission critical.
- Small Memory leaks may not cause major problems.
  - PC's used to browse the web have quite large memories.
  - The typical use of web application does not last very long.

The current situation is making many of the previously listed reasons obsolete, thus further underlining the importance of the problem at hand.

- Web applications with thick clients are currently widely built and used.
- Web engineering is maturing.
- Web is such an interesting platform, that it is gaining interest also from mission critical application developers.
- Web applications are run on diverse platforms, on which some are memory critical.

Web applications with thick clients are currently being widely used and developed. A concrete example of this trend is the rise of Chromebooks. During 2014's third quarter, Chromebooks outsold iPads in the US for education purposes [9]. Chromebooks operating system is Chrome OS, on which applications are actually web applications. The maturing of web engineering can be seen by looking at the adoption of the technology (for example *node.js* gaining popularity), and improvements in the tools which embrace the technological foundations of the web, for example: build tools (gulp[2], grunt[3]), linting tools (jsLint[4], jsHint[5], eslint[6], jscs[7]), package managers

[2] `http://gulpjs.com/`
[3] `http://gruntjs.com/`
[4] `http://www.jslint.com/`
[5] `http://jshint.com/`
[6] `http://eslint.org/`
[7] `http://jscs.info/`

(npm[8], jspm[9]), and test tools (mocha[10] , phantomjs[11]).

3.4 ECMAScript 2015, other standards and tools

In this section we will look at the newest edition of the *ECMA-262* standard, which is *ECMAScript 2015* [10], we also take a look at other standards, including *Web Components* [84] and also few tools. We take a look at these, focusing especially on the features related to the development of mission critical, memory leak free web applications.

3.4.1 ECMAScript 2015 and Memory Management

The newest edition of the ECMA-262 standard is ECMAScript 2015 (*ES6*, *Harmony*, *ES.next*), the standard is currently under development, but it is allready feature complete [27]. From JavaScript's memory management point of view, the standard contains some interesting features. The standard also takes into account the languages misunderstood features, namely prototypal inheritance. In this subsection we go through the updates in the standard, related to languages memory management capabilities.

JavaScript has had only function scoped variables, but the edition 6 introduces the keyword **let**, which makes it possible to declare block scoped variables. Eventually let will replace var's.

As we have concluded in the earlier sections, JavaScript's memory leaks are logical programming errors, caused by uncleared references. The new edition includes a possibility of creating weak references, with the addition of **WeakMap** and **WeakSet**. WeakMaps contain key value pairs, on which the keys of the map are of the type Object only [25]. WeakSet objects are a collection of weakly held objects [26]. Weak references mean that if an object has only one reference, and the reference is weak, the object referenced by it, is eligible for garbage collection.

Previously JavaScript has not had a built-in support for modules. Allthough the JavaScript community has developed elaborate workarounds, which have progressed into two separate standards. These module standards for JavaScript are; *CommonJS* (CJS) [80], and *Asynchronous Module Definition* (AMD) [81]. Unfortunately these

[8] `https://www.npmjs.com/`
[9] `http://jspm.io/`
[10] `http://mochajs.org/`
[11] `http://phantomjs.org/`

two standards are incompatible. CommonJS modules are widely used on the server-side, i.e. in *node.js*. Asynchronous Module Definition's most popular implementation is *RequireJS* module loader [82]. The newest edition of ECMAScript adds a native support for modules. Modules improve JavaScript in general greatly, and they are related to memory safety.

JavaScript's prototypal inheritance has been largely misunderstood by developers coming from other more "traditional" object-oriented languages, like *C++*, *C#*, and *Java* [8]. The newest edition of ECMAScript brings *classes*, and inheritance via *extends* into JavaScript, making the language easier for developers coming from these kinds of programming languages.

3.4.2 Other Standards and Tools

Many current browsers support some features of the ECMAScript 2015 already, but in addition there are transpilers, like *babel*[12] and compilers, such as *traceur*[13] which make it possible to currently take advantage of even more ECMAScript 2015's features.

Dynamically typed languages have been claimed to be unreliable, since errors can be caught at runtime instead of during compilation. Currently there are tools for developing JavaScript with support for static typing. One such tool being *TypeScript*, which is a typed superset of JavaScript, that compiles into plain JavaScript [76]. There is also another tool called *Flow*, which takes a different approach to static typing, bringing static type checking into JavaScript [13]. Static type checking warns the developer, when the program modifies objects structure during the runtime.

Web Components are a set of technologies, trying to improve the modularity of web applications. Shadow DOM is an upcoming standard, related to Web Components. The Shadow DOM standard makes it possible to encapsulate parts of the DOM [85]. Improving the modularity is in general a good idea, and DOM has been previously criticized by basically being a global variable.

[12] `https://babeljs.io/`
[13] `https://github.com/google/traceur-compiler`

4. MANAGING AND DIAGNOSING WEB APPLICATION'S MEMORY

In this chapter we consider what it takes to develop applications that do not contain memory leaks. As application's complexity grows, their tendency to contain logical programming errors grows with it. This is the reason why debugging and diagnosis are needed. Detecting memory leaks can be very difficult and may require deep insight of the application, since the problem of deciding after what point an object is no longer used is *undecidable* [34]. To identify the objects that will not be needed, requires knowledge of the application's flow and structure. Processes have been proposed to be used, when identifying leaks from web applications, these are usually carried out in the development phase, with the use of browser's developer tools.

4.1 Diagnosing memory consumption with developer tools

Nowadays the most popular desktop web browsers are shipped with built-in Developer Tools. These are the front-end developer's main tools for inspecting the application instances behaviour in a live environment. They enable the developer to work with the client-side technologies (described in section 2.3). Developer Tools, like browsers in general, have evolved convergently. Their features are not standardized in any way, and the features can be seen to have evolved (more or less) around developers needs.

In this section we'll look into analysing web applications memory usage with the use of Google Chrome's Developer Tools. Google Chrome's Developer Tools are actually based n WebKit/Safari's Web Inspector. Chrome's Developer Tools were selected since they currently provide the most advanced features regarding web application's memory usage debugging. The most important features Chrome's Developer Tools have, considering memory usage are: timeline recording, heap snapshots, forced garbage collection, and object allocation tracker. Google provides quite a comprehensive guide on profiling web application's memory usage with the Chrome's Developer Tools [19]. This article explains how these these tools can be used, here we will discuss on the usefull features these tools have, regarding application's memory

analysis.

A. Osmani suggests the following process for identifying memory leaks from web applications, with the use of Google Chrome's Developer Tools, in his presentation Memory Management Masterclass: [41]

1. Check Chrome Task Manager, if applications memory usage is growing.
2. Identify the Sequence that is suspected to be leaking.
3. Do a Timeline recording and perform those actions.
4. Before performing the actions, force Garbage Collection.
5. Analyze the timeline data, sawtooth curve implicates that lots of short lived objects is allocated.
6. Use the object allocation tracker to narrow down the cause of leak.

In the 1st step a Google Chrome's tool called Task Manager is used, it has similar characteristics when compared to for example Windows' Task Manager. Task Manager can be used to inspect browser tabs; memory usage, JavaScript memory usage, GPU's memory usage, CPU usage, and more. Figure 4.1 shows a screenshot of Google Chrome's Task Manager.

In Figure 4.2 is presented Google Chrome's (*version 41.0*) Developer Tools' Timeline view. The Timeline view makes it possible to record runtime environment's behaviour during its use, and afterwards to inspect the recording. It does not only show the heap's size, but at what moment garbage was collected, and how much of it was found and released. This view is used during 3rd, 4th and 5 steps in the aforementioned process. Regarding memory analysis the most interesting information the timeline presentation has to offer are: JavaScript engine's heap's size, amount of DOM nodes, DOM event listeners, and the Garbage Collection Events.

It is also possible to *force garbage collection* from timeline view, which is usefull before starting and ending the recording. Timeline also makes it possible to export the timeline data in *JSON* format. This makes it possible for example tester to do timeline recording, and send the timeline recording JSON file for the developer who can then analyse it. This JSON could also be further analyzed automatically.

Google's Developer Tools also have a Profile-view, which enables the analysis of heap's contents by enabling Heap Snapshots to be taken and analysed. Before Chrome takes the Heap Snapshot, it automatically does garbage collection, which makes

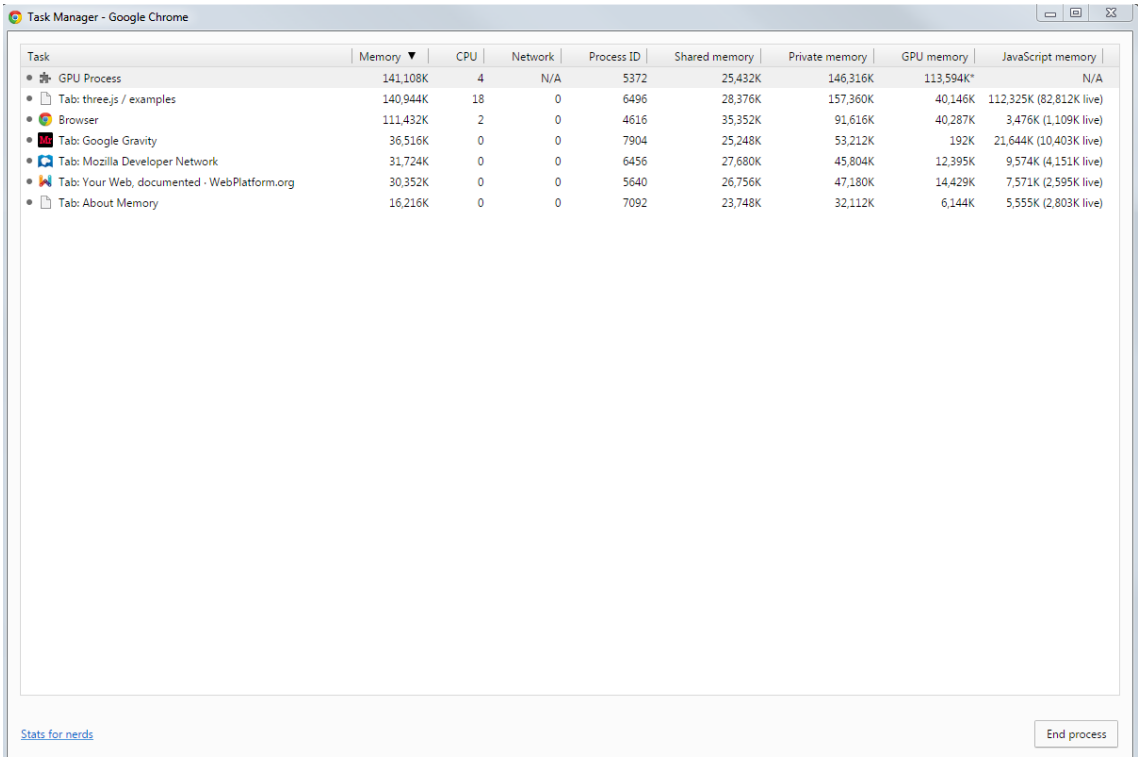

Figure 4.1 *Google Chrome's Task Manager*

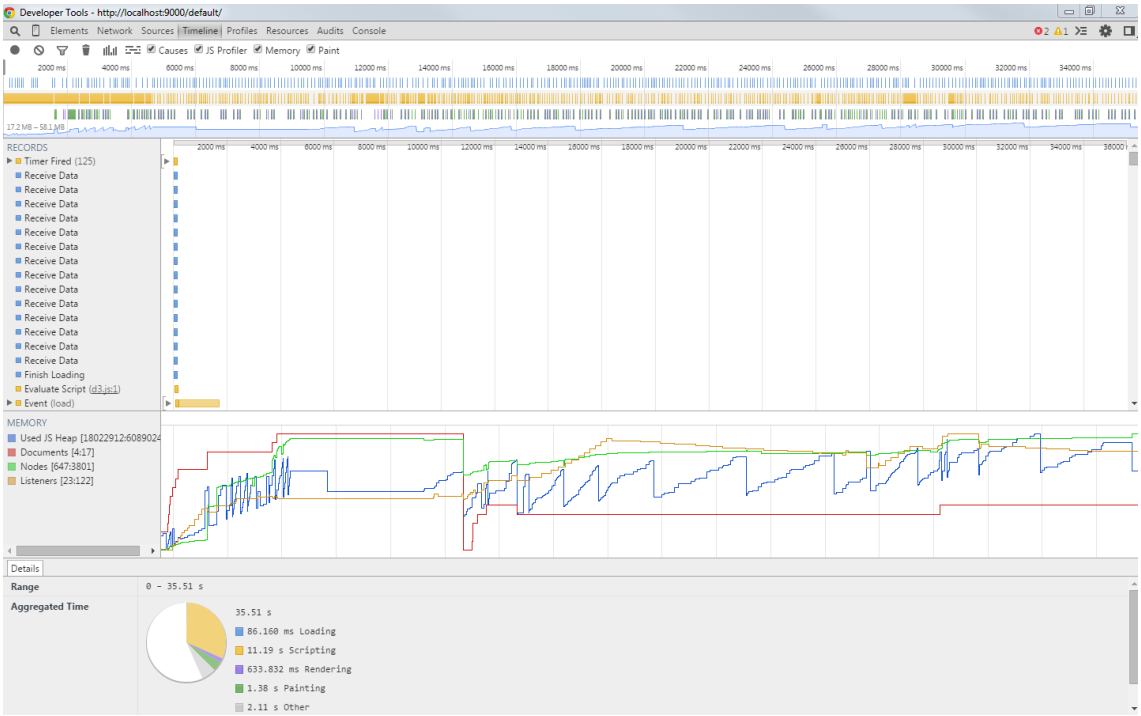

Figure 4.2 *Google Chrome's Developer Tools Timeline view*

the snapshot's contents more easily interpreted. The Profile-view also enables Heap

allocations to be recorded, with a tool called Object Allocation Tracker. The Profile-view is presented in screenshot in Figure 4.3. This is a valuable tool for isolating memory leaks.

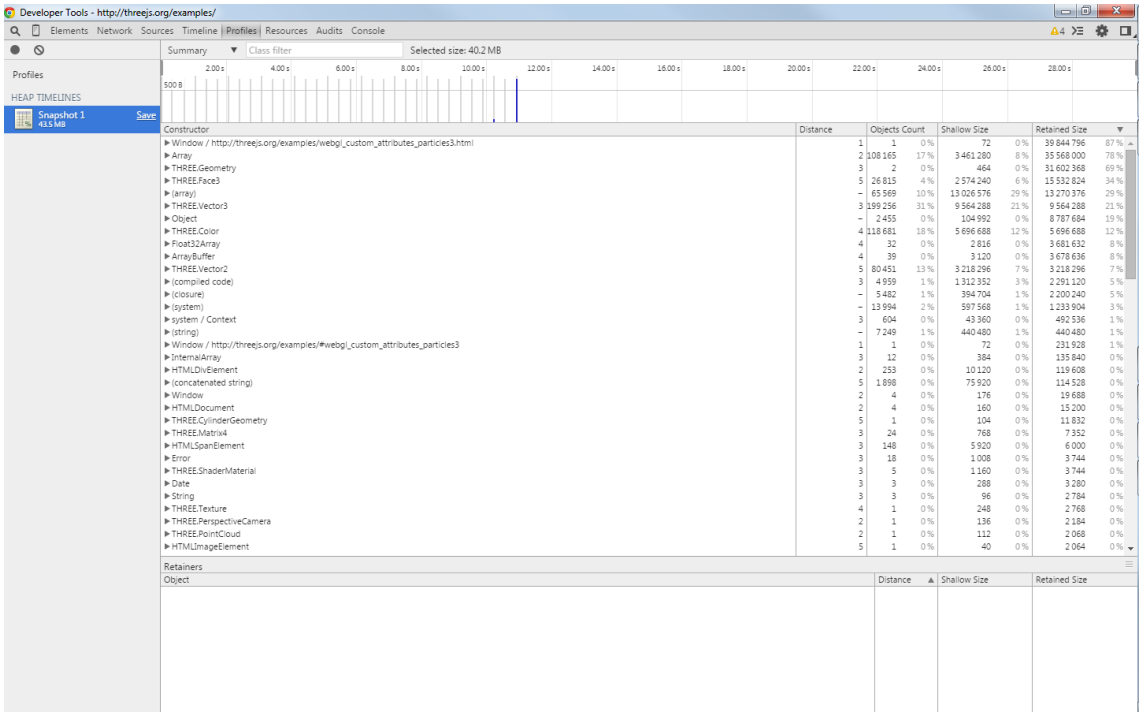

Figure 4.3 *Google Chrome's Developer Tools Object Allocation Tracker*

The Profile view gives the user four possible views for inspecting the heap's contents: Statistics, Summary, Containment and Comparison. The Statistics view provides a quick overview, on what are stored in the heap: Code, Strings, JS Arrays, Typed Arrays, System Objects.

Summary view shows heap's content categorized by the objects' constructors. It also shows the amount of allocated objects, their retained size, and shallow size. The Profiler's window's lower part shows what is the retaining path of the selected object, therefore illustrating what is preventing the object from being removed. Heap Snapshots and the way they are presented in Developer Tools are useful when identifying possible DOM leaks. Profiler highlights the DOM nodes that are currently detached from the DOM tree with red colour. The profiler highlights the DOM nodes that have direct references from JavaScript objects with yellow. So the way to find the cause of the detached DOM tree leak is to: identify the red node, look for the yellow node on its retaining path, and then identify the JavaScript reference.

Comparison view lets the user to compare snapshots content with each other. Making it possible to analyze what objects have been removed, and what have been allocated

between snapshots. This enables the user to confirm that memory was correctly released, or what causes the memory not to be released (retaining path).

Containment view shows the JavaScript engine's Heap's contents, in a hierarchical fashion. Letting the user to inspect the Heap's and objects' structure. The Figure 4.4 illustrates how Containment view presents a situation where, an HTML button's reflector's (`btn`) lexical scope contains a reference into `largeString` variable.

```
var returnEventHandlerAndDoStuff = (function () {
    var largeString = new Array(1000000).join("x");
    var doSomethingWithString = function () {
        largeString += "x";
    }

    var evHandler = function () {
        console.log("hi, from evHandler");
    }

    return evHandler;
}());

function init() {
    var btn = document.getElementById("btn");
    btn.addEventListener("click",
        returnEventHandlerAndDoStuff, false);
}

init();
```

4.2 Avoiding Memory issues with development practices

So as we have concluded, there is no *fundamental reason* why web applications should contain memory leaks. Therefore it is perfectly feasible to create memory leak free web applications. This can, and should be tackled by using and enforcing good development practices, since memory leaks are allways caused by logical programming errors. In this section we describe good development practices in using memory in such manner that Garbage Collector can do its job effectively.

The most important development practices in avoiding memory leaks, are very well known and widely used, in all the major programming languages. Here we list some development practices that are related to memory management. For more comprehensive JavaScript coding convention guide, see D. Crockford's *Code Conventions for the JavaScript Programming Language* [4].

Figure 4.4 *Google Chrome's Developer Tools Profiler, Heap Snapshot Containment-View*

Be carefull when declaring variables. Declare variables *explictly* as otherwise JavaScript will create *implied global*, when *strict mode* [30] is not used. Declare variables *using appropriate variable scope*, making sure the scope of the variable is *as minimal as possible*. This can also be seen as enforcing *the principle of least priviledged*. ECMAScript 2015 will make defining block scoped variables possible, with the *let*-keyword. Block scoped variables should be taken into use as soon as possible. Aventually they should be used instead of var's altogether.

Global variables should be avoided for multiple reasons, and memory management is one of these. Browser's JavaScript engine usually cleans the global variables when

new page is loaded, or current page is refreshed. In Single Page Applications, the interval between page loads has been reduced by design. This makes it possible for global variables to accumulate over time.

Application's **JavaScript code should not hold references to DOM nodes for extended periods of time**. Long lasting references to DOM nodes can lead to leaking memory in such a way that after the node is removed from the DOM tree it cannot be removed from the memory, since it has a live reference to it. In Section 3.3, we described such a leak, and in the following example we show how it could be avoided by explicitly setting the DOM object's reference to null.

```
lightController.view = document.createElement('div');
lightControllerCollection.appendChild(lightController.view);

....

lightControllerCollection.removeAllChildren();
lightController = null;
```

So if a variable will not go out of its scope, after its intended use, the variable should be *dereferenced/nullified*) to enable its garbage collection. When dealing with references to DOM nodes, the developer should be carefull, in order to enable DOM tree's effective garbage collection. In a similar fashion as variables that are not going to be used, `eventListener`s and `setInterval`s that are no longer needed should be removed and/or cleared. Thus every `setInterval` and `addEventListener` should have corresponding `clearInterval` or `removeEventListener`.

Events are also widely used within SPA Frameworks, for example to synchronize models with views. SPA Frameworks have different approaches to memory management. In Subsection 2.4.4 we mentioned SPA frameworks AngularJS and BacboneJS. AngularJS takes a more holistic approach to applications architecture and structure, Backbone on the other hand gives the developer more freedom. This means that Backbone for example does not have view-lifecycle management, which makes Backbone's state changes prone to memory leaks. The developer has to clean the views and unbind their events explicitly. This can lead to memory leaks, for example if the developer removes a view, which has registered to be notified about models changes. This means that the model has a reference to the view's callback function, preventing the view to be removed from the memory, until the developer explicitly unregisters the event listener, even though it has been detached from the DOM. This is infact an instance of the *Observer pattern*'s *lapsed listener problem* [72]. These could be solved by weak references, as we will discuss in Section 5.2.

As we mentioned earlier JavaScript's prototypal inheritance is different than what developers coming from class-based languages are used to. When doing inheritance **Appending object's methods to the prototype**, instead of the object in constructors can help to reduce applications memory footprint considerably. If the objects methods were assigned to the object (not the prototype), new copy of the function would be allocated every time, new object would be created. Since the duration of garbage collection cycle is related to the memory footprint, this is beneficial also for application's performance.

```
function Fellow (name) {
  this.name = name;
}

Fellow.prototype.changeName = function (newname) {
  this.name = newname;
}

var f = new Fellow("fry");
f.changeName("zoidberg");
```

As described in Subsection 2.3.3, **closures** make it easy for developers to obfuscate reclaimed memory. Google's article *Optimizing JavaScript* claims that closures are *"the most common source of memory leaks"* [17]. In the following example we show an example how to leak memory with closures. As long as there is a reference to inner-function, the `largeString` in the following example cannot be Garbage Collected.

```
function outer() {
  var largeString = new Array(1000000).join("x");
  // do something with the largeString

  function inner() {
    // do not do anything with largeString
  }

  return inner;
}
```

If `largeString` is only used in the `outer` function, it should be *nullified* after its use, to make it possible for the garbage collector to deallocate its memory (or rather the use of a closure should be re-considered in this use case). However the current implementations of v8 and Spidermonkey identify the above example's closure variable (`largeString`) as garbage. These are however JavaScript engine optimizations, and

developers should not start to rely on these optimizations to provide functionality. Instead developers should understand the language's important features, such as closures in order to be able to use them apropriately. For example these optimizations do not recognize `largeString` variable as garbage in the following example. Similar issue was discovered from the Meteor framework's source code [14].

```
function outer() {
  var largeString = new Array(1000000).join("x");

  (function () {
    largeString += "x";
  }());

  function inner() {
    // does not use largeString
  }

  return inner;
}
```

Callbacks, closures, and anonymous functions are JavaScript's characteristic features and it is important to understand their relation to memory management. Developers should **pay attention at what scope functions are declared in**. For example, when the code uses the same anonymous function multiple times, JavaScript engine creates new instance of the function every time. Instead of using an instance of a function. This has negative effect on memory usage and thus performance. The following example demonstrates, a closure being created (possibly accidentally), and an instance of anonymous function being created for every eventListener.

```
function addEventListeners() {
  var els = document.getElementsByTagName("td");
  for (var i = 0; i < els.length; i += 1) {
    els[i].addEventListener('click', function () {
      this.style.color = 'red';
    }, false);
  }
}
```

The anonymous function's lexical scope, maintains references to the scope's variables (in this case for example variable `i` and list `els`). This in turn also creates circular references, between the DOM element's `td` event handler's lexical scope, which references into the Element list (`els`), which references into the DOM Element. This does not however itself leak any memory on current browsers, but it rather underlines the point of understanding and using anonymous functions, and closures for

what they are really usefull for. A better way to achieve the same functionality would be the following.

```
function changeToRed() {
  this.style.color = 'red';
}

function addEventListeners() {
  var els = document.getElementsByTagName("td");
  for (var i = 0; i < els.length; ++i) {
    els[i].addEventListener('click', changeToRed, false);
  }
}
```

It has been already stated, that JavaScript engines' Garbage Collection is triggered by memory allocation. However JavaScript does have *delete* keyword, which might be confusing: it does not deallocate the memory used by entire objects, but it is intended for modifying object's structure by removing properties during runtime. However with current JavaScript engines implementations (at least in v8), *delete*-keyword's usage is not advised. [41]

4.3 Runtime Memory diagnostics

If web technologies were to be used in creating large mission critical web applications, the use of best practices can become quite difficult to oversee. Therefore safety nets should be put in place. This section studies two alternatives on programmically diagnosing applications memory usage in the runtime environment, in order to diagnose applications memory usage and to regognize and isolate memory leaks.

4.3.1 Browser extension

One way for diagnosing web applications memory consumption would be to build browser extension. Browser Extensions have lower level APIs when communicating with the browser's runtime environment. For example browser extensions can have special bindings for the browser's developer tools, thus enabling for example heap snapshots to be taken programmitically.

There are advantages and downsides in using browser extensions for doing runtime diagnostics of the web application. The advantages are the additional features enabled by closer integration to the web browser. The downsides of this approach, go

against the motivators that were listed in Section 2.4.1 regarding why the web platform has been considered so intriguing. One disadvantage is that browser extensions are not standardized (*Open Application Formats*), which means that the extensions should be built separately on different browsers. Platform independecy was listed earlier as one of the advantages of building web applications, and building browser specific extensions to extend web applications features would be going against that notion. Another downside of building a browser extension for diagnosing purposes is that the extensions have to be explicitly installed, the lack of installation step, was also considered as one of the advantages of building native web applications. The separation of diagnostics from the application adds maintenance work and a new paradigm to the application as a whole.

4.3.2 performance.memory API

Google's GMail team have had experiences in debugging their Single Page Application's memory issues with runtime diagnostics. When doing so *Google Chrome* introduced an API for the web applications to read JavaScript engine's memory usage data from the running client-side application. It should be noted that the *performance.memory* API, that was introduced into Chrome, is not currently part of any official standard or a candidate into becoming a standard. However it has been used successfully when diagnosing GMail's memory issues. This API was also used succesfully during the work on this thesis when creating our demo application.[54]

With the use of this API, application developer has the ability to inspect how their application is using memory during its use. For example in comparison to Heap Snapshots the implementation does not provide very specific data. Google Chrome's API exposes new values of JavaScript Engines memory usage every 20 minutes, however in the default mode the values are not accurate. The values exposed by this API are quantized to specific buckets, so that the values are not as refined when the values are large. This is done in order to reduce the risk of the information's misuse, as it has been speculated if the data could be for example used in Side Channel Attacks. However when starting Google Chrome with a flag *–enable-precise-memory-info*, this API provides real values in real time. Which make the API more useful when diagnosing memory issues. [61]

The *performance.memory* API returns three byte values about applications memory consumption:

1. **jsHeapSizeLimit** - JavaScript engine's heap's size-limit.

2. **totalJSHeapSize** - Allocated memory for JavaScript engine's heap (including free space).

3. **usedJSHeapSize** - Memory currently being used.

During the work on this thesis we developed a JavaScript module, which we used in our demo application's memory usage diagnostics. The module can be configured to read `performance.memory`'s values at a given interval. Choosing the appropriate interval is a trade-off between causing overhead for the application and getting usefull data. In runtime diagnostics we are actually most interested in detecting the frequency of garbage collecions, how much memory was collected and what is the heap's size after the collection. In order to ensure that the heap's size is not growing unnecessarily and that the garbage collector is not impeding with the application's interactivity.

During this thesis we used the *performance.memory* API to deduce approximations of garbage collection stops, how much was collected, and what is the heap's size after the collection. In order to do this, we read samples from the memory usage values. And calculated the change between the heap's size, in order to find out garbage collection stops. Necessarily not all garbage collection stops can be identified, only by observing the JavaScript engine's heap's size: garbage collection is usually triggered by memory allocation, so in case when the newly allocated memory is larger than the amount of garbage, there is no way of knowing that garbage collection took place. The following code example shows the basic functionality of sampling performance.memory's `usedJSHeapSize`, to detect garbage collection stops and their characteristics.

```
var diagnoseMemory = (function () {
    var prevHeapSize
        = window.performance.memory.usedJSHeapSize;
    var prevGCTimeStamp = Date.now();

    return function() {
        var mem, delta, cycle;
        mem = window.performance.memory;
        delta = mem.usedJSHeapSize - prevHeapSize;

        if (delta < 0) {
            console.log("GC: " +
                Math.abs(delta) / 1000000 + " MB");
            console.log("Heap: " +
                mem.usedJSHeapSize / 1000000 + " MB");
            cycle = Date.now() - prevGCTimeStamp;
```

```
            console.log("Cycle Length: "
                + (cycle / 1000) + " seconds");
            prevGCTimeStamp = Date.now();
        }

        prevHeapSize =  mem.usedJSHeapSize;
    }
}());

setInterval(diagnoseMemory, 50);
```

In this example, we use an interval of 50 milliseconds, which did not cause noticable overhead on our test environment. It should be noted that the `delta` and `cycle` values are approximations, since memory allocation has taken place in between the samples. The previously presented function provides the basic functionality, but in order to do something usefull with the information the data needs to be stored for further analysis.

If web workers are used, it should be noted that browsers may spawn new process for web workers. There are two types of web workers: Dedicated and Shared. In Chrome / Chromium, dedicated workers run in the same process as its parent render process, but in their own thread. Shared worker runs in its own separate process called worker process, and its connected to multiple render processes via the browser process (hence the name). So if dedicated workers are used, their memory footprint is included in the performance.memory APIs results.

Node.js has `process.memoryUsage()` API [39], which returns similar object as `performance.memory`. Thus this API can be used similarly in server side runtime memory diagnosis. The memoryUsage method returns an object containing the following values:

- **rss** - Resident Set Size, the portion of the process's memory held in RAM [75].
- **heapTotal** - Allocated memory for JavaScript engine's heap, including free space.
- **heapUsed** - Memory currently being used.

These two values, `heapTotal` and `heapUsed` are similar to performance.memory's `totalJSHeapSize` and `usedJSHeapSize`. When comparing these API's together the value that is missing is `jsHeapSizeLimit`, which has been replaced by the `rss` value.

5. EVALUATION

Understanding application's memory consumption during runtime can be very difficult, especially when considering rich SPA applications that may have very complex state.

Memory leaks are logical programming errors in the applications, therefore many memory leaks could be tackled by using development practices listed in the Section 4.2. The first step we recommend during the development phase is to use a linting tool. Linter should be used to enforce good coding conventions, and to warn about possible programming errors that could cause memory issues. Code reviews are a valid practice for multiple reasons, including identifying logical programming errors such as dangling references.

By code reviews and testing we can currently confirm possible memory issues, we cannot however be certain that the application is free of such issues. This being the reason why it can be useful to continuously diagnose applications memory consumption in the runtime, in order to fix occuring problems quicker. When applications have complex states', complex sequence of actions can lead the application into state where some of its memory will not be released, even though it was supposed to. Those logical errors in the application could be located and fixed more easily if proper diagnostics where to be at place.

If the `performance.memory` API were a standard, and by default gave real values, runtime memory diagnostics could be used to catch possible memory issues as soon as they arise on client's application instance. Enabling identifying of memory issues that did not arise on explicitly defined test cases, or during experimental testing. As that is not currently the situation, more feasible solution would be to run diagnostics as part of release candidate environment, where testers and developers are told to run their browsers with the `-enable-precise-memory-info` flag. This would enable centralized collection of memory diagnostics data during development and testing. In the following sections we evaluate the usefulness diagnostics module's developed during the work on this thesis.

5.1 Results

Our criteria for accepting our diagnostics is to confirm that with our runtime diagnostics we are able to identify possible memory issues. We confirmed its function by diagnosing our demo application's memory usage for over a day, exported the diagnostics data, analyzed it and used it to identify possible memory issues.

During the work on this thesis we used the `performance.memory` API with Chrome started with `-enable-precise-memory-info` -flag to perform JavaScript runtime's memory diagnostics. Our demo application included the diagnostics, which performed diagnosis on the background of the application. We left the application running for over a day, and collected diagnostics data of this session. After that we inspected the data to confirm possible memory issues.

We modified the previously presented diagnostics code, so that, instead of logging the values into console, they are stored in the browser's *indexedDB* [63]. From which we exported the JSON data after the test period. We then analyzed the data and calcucated memory usage's characteristic values of the analysis session.

After recording diagnosis data of our demo application for the given period, we exported the data. Then we calculated characteristic values for the session's memory usage. These characteristic values include garbage collection frequency, garbage collection amounts, garbage collection rate and heap's size (see Table 5.1).

	Heap Size (MB)
avg	29,04
min	17,70
max	71,50

	Garbage Collection Amount (MB)
avg	5,48
min	0,0001
max	51,56

	Garbage Collection Cycle (S)
avg	2,89
min	0,05
max	6,12

	Garbage Collection rate (MB/S)
avg	1,90

	Duration
Seconds	85574,99
Hours	23,77

	Amount Of Garbage Collections
#	29640,00

Table 5.1 *Heap usage characteristics*

We plotted a graph of the application's heap size after garbage collection over time (see Figure 5.1). In this graph is also presented the garbage collection amount and JavaScript engines heap's total size over time. From this graph we noticed that the application's heap size is not constantly growing. However it indicates that many short lived objects are constantly being allocated, generating lots of garbage, making the garbage collection stops frequent.

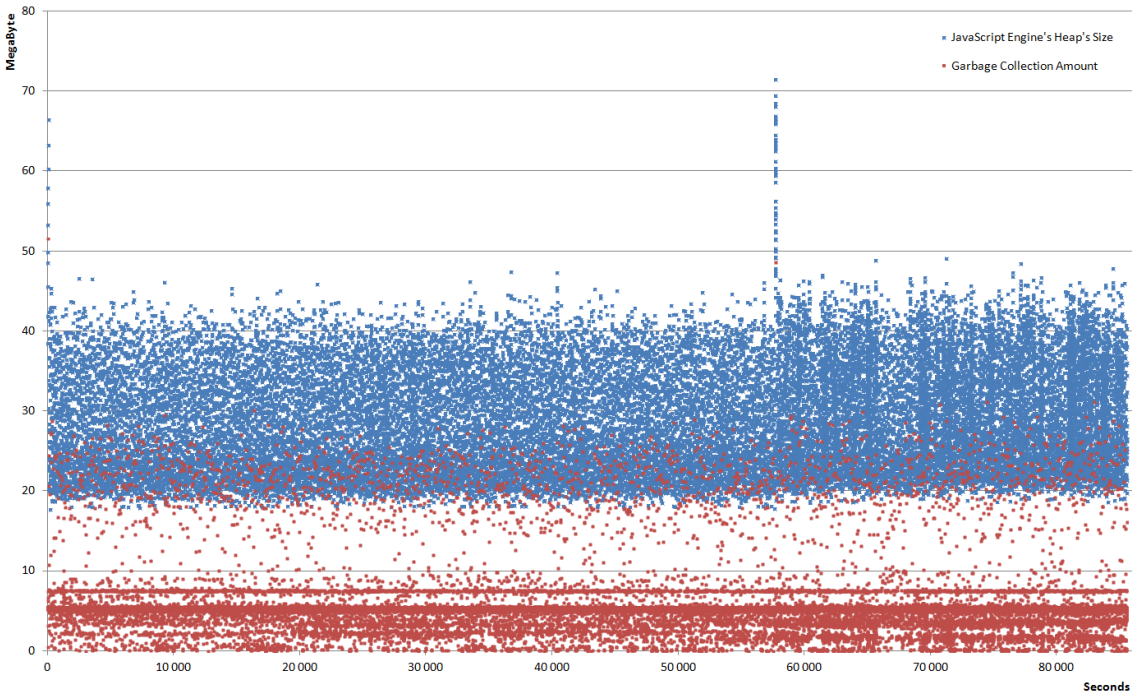

Figure 5.1 *Heap usage and garbage collection over time*

This information can be compared to the information what could have been gotten with Developer Tools' Timeline view. However Timeline view contains more information that we would also find usefull, for example the amount of DOM nodes and documents. However implementing diagnostics as part of the application has several benefits. The main difference being, that the diagnosis happens on the background. Another advantage is that timeline recordings are not feasible for diagnosing application's usage on extended periods.

5.2 Open Issues

The `performance.memory` API is not very capable in isolating possible DOM leaks, since only the DOM nodes wrappers are stored in the JavaScript engine's heap, not the DOM nodes themselves. Therefore leaked DOM nodes reflect on the heap's size only deliberately. For diagnosing DOM leaks, it would be helpful to diagnose the amount of DOM nodes, even more so how many of these are part of the DOM tree. If a DOM node is not part of the DOM tree and is still stored in browsers memory, it has not been removed from browser's memory because, either JavaScript code is still holding references to this object, or the browser has not yet had the time to remove it. More profound question underlying might be, should the DOM nodes and JavaScript objects be managed by the same garbage collector. Mozilla's Servo web browser engine has taken this stance [36].

WeakMaps and WeakSets cover some use cases for weak references, but not all of them, most importantly the use case that cannot be covered by them, is possible memory leaks caused by the *Observer pattern* [71] [12]. The Observer pattern is a software design patterns, in which a Subject holds references to its observers, notifying them automatically of state changes. Memory leaks caused by the Observer pattern are also called Lapsed listener problem [72]. These kinds of leaks happen when the listener (observer) forgets to unsubscribe from the subject, after it is done listening. The subject maintains a reference to the observer, preventing the observer's garbage collection. Observer patterns are widely used in Model View Controllers (MVC), in synchronizing the views with the models. In this use case the views being the listeners and models the subjects.

By holding weak references to the observers, the lapsed listener problem could be avoided. There is an unaproved, preliminary, proposal of weak references, in the ES-Harmony's wiki page (language beyond ES6) [12]. However there has been some opposition to the idea, mainly because it ties weak reference's behaviour to garbage collectors non-deterministic behaviour.

Fortunately Chrome's Developer Tools are quite usefull when detecting DOM leaks. As it happens, in client side application's lapsed listener problems are quite often related to DOM leaks, because the views in web applications (listener) are presented through the DOM. If the view does not unsubscribe from model's updates, the model maintains a reference to the view, which prevents the view from being removed. This results in leaked DOM nodes, presuming that the view has removed itself from the DOM tree.

Memory profiling tools have improved lately in all the major browsers: Internet Explorer 11 and Firefox also have the ability to do timeline recordings and to take Heap Snapshots [20] [3].

In this thesis we presented the diagnostics part of the memory profiling. Further actions would be to investigate into the analytics. The values provided by our diagnostics could be send to the server, and they could be bind to the user session. We could for example analyze how much memory does the application use on average. This in turn could be used to catch regression in application's memory usage, and make the memory usage visible for the developers. Data collected from individual sessions could be used to analyze applications usage in more detail, for example does the applications memory usage grow signifigantly over the session. If the data is bind to certain context (open view/components, etc.), the data can be used in isolating memory leaks.

During this thesis we confirmed some memory issues in our demo application, that require further investigation. The next step would be to investigate what is generating the garbage, this can be done by profiling the application's code and by identifying frequently executed functions. After that, these functions should be examined more closely: does these functions allocate large amounts of memory unnecessarily, for example, do they contain many function or object definitions.

Diagnostics could be used for example so that, we enable diagnostics during experimental testing, and make sure that the testers start their browsers with the proper flag. This would make it possible to automatically gather the clients memory usage characteristics during experimental testing period. This API can also be used in automatic testing, for example to catch regression in application's memory usage.

Some kind of heuristics could also be implemented on top of the diagnostics, for example, to identify situations where applications heap's size is constantly growing as memory leaks.

5.3 Discussion

Garbage Collection and memory management are very intricate tasks, and we think that in general the current runtimes are performing well, but improving the visibility to runtime environment would be helpful. Mainly because currently it is quite easy to create applications that do not handle their memory allocations. ECMAScript standard does not take a stand about the implementation's garbage collection. However developers usually have certain expectations about garbage collection, when they are developing applications. Causing an impedance mismatch, between the web platform and the expectations of developers.

Currently the garbage collector has been fully transparent, meaning that it is trying to provide full abstraction to an underlying undecidable problem. From application developer's point of view currently garbage collectors have been fully abstracted, and their task has been to prevent *out of memory* situations. Exposing parts of the garbage collector for the application developer would force to rethink garbage collector's semantics. However trying to provide full abstraction to an undecidable problem is deemed to fail.

If some API's would expose garbage collectors behaviour to the application developer, their role could shift towards resource management framework. Which in turn could possibly constrain the development of the garbage collectors. Possible upcoming features regarding to JavaScript engine's memory management, should most

likely be designed in such a way that complexity is not added to the garbage collectors itself.

The dialog about Garbage Collector's role is timely, as there is allready discussion going on about the upcoming ECMAScript 7's proposed introduction of weak references into the language, which has also started the conversation about the role of Garbage Collector and its semantics [11].

When comparing browsers to other runtime environments, like *Apache Flex*, and *Java Virtual Machine (JVM)*, browsers' JavaScript runtime gives the application developer the least control and visibility over managing and diagnosing the runtimes memory. Apache Flex has an API for forcing garbage collection (`System.pauseForGCIfCollectionImminent`), which is intended for interactive applications (in avoiding the embarassing stops, eg. during audio / video). Java has an API's for suggesting the virtual machine to do garbage collection (`System.gc()`). These could possibly be useful when developing interactive applications on the browser. But in the scope of this thesis we are not as much interested as those, as we are in diagnosing the applications memory usage; heap's size, amount of DOM nodes, and garbage collection cycles. The Java Virtual Machine (JVM) has an API, that would be usefull for our use case; JVM's `GarbageCollectionNotificationInfo`. This API provides an event when Garbage Collection happens and provides basic information on the heap's usage. This helps in diagnosing the memory usage of the application with an appropriate abstraction, investigate applications memory usage and to identify possible memory issues. During this thesis we went around the web platforms limitations to achieve similar functionality provided by this API. If there were this kind of API, it would remove the need to continuously sample the heap's usage in order to identify garbage collection stops. Which would in turn reduce the overhead caused by our runtime memory diagnostics.

W3C has formed a *Web Performance Working Group* [86], which is part of the *Rich Web Client Activity* [87]. Its mission is to provide methods to measure aspects of application performance of user agent features and APIs. That being said, W3C is commited to enabling the development of performant web applications. This could mean that if the security concerns of exposing the memory usage data, are evaluated to be non existent, then *performance.memory* API could possibly become a standard. Further analysis will be needed on possible disadvantages of exposing runtime's memory usage diagnostics to the client.

6. SUMMARY

The problem of deciding when memory will no longer be used is undecidable, therefore there will allways remain cases where in the object's lifecycle, a developer needs to explicitly make the object eligble for garbage collection. It might be complicated to identify such issues, which is where proper diagnostics could help to steer the developers into the right direction.

Another reason being that as the browser is turning into ever more capable runtime environment, ever more complex applications are going to be built on top of it. The probability of logical programming errors is growing along with client side applications' complexity. Improved visibility into the runtime would help developers to be informed about such issues and fix occuring problems as they arise.

By using the diagnostics module developed during this thesis, we received results that enable us to diagnose the application's memory usage over the user session. This data can then be used to define characteristic values of the application's memory usage. These characteristic values are: garbage collection rate, garbage collection cycle's length and amount of garbage collected. These values improve the understanding of application's memory usage. With the use of our diagnostics module, we were able to establish a baseline for our demo application's memory usage. Based on this it would be easy to detect applications regression in memory usage during the development. The method the diagnostics was achieved is far from perfect. Because it relies on Chrome's proprietary API, which in turn is constantly being polled in order to detect garbage collection stops. But currently this is the only way for the application to read any information about browsers memory usage and to recognize the garbage collection stops.

Web applications are often compared to native binary applications, and often times they are called in being inferior in performance. As we have concluded memory management and application's performance are closely related, and in order to improve web application's performance, improved visibility to the runtime would be helpfull in understanding and fixing these problems.

BIBLIOGRAPHY

[1] T. Berners-Lee, Long Live The Web, Scientific American

[2] T. Berners-Lee, The WorldWideWeb browser, [WWW], Available: http://www.w3.org/People/Berners-Lee/WorldWideWeb.html Accessed 10-January-2015.

[3] D. Camp, D. Callahan, Developer Edition 44: New visual editing and memory management tools, [WWW], Available, https://hacks.mozilla.org/2015/11/developer-edition-44-creative-tools-and-more/ Accessed 15-March-2015.

[4] D. Crockford, Code Conventions for the JavaScript Programming Language, Mar 2015, [WWW], Available: http://javascript.crockford.com/code.html Accessed 10-January-2015.

[5] D. Crockford, The World's Most Misunderstood Programming Language, Mar 2015, [WWW], Available: http://javascript.crockford.com/javascript.html Accessed 10-January-2015.

[6] JSLint, The JavaScript Code Quality Tool Mar 2015, [WWW], Available: http://jslint.com/ Accessed 10-January-2015.

[7] D. Crockford, JScript Memory Leaks, [WWW], Available: http://javascript.crockford.com/memory/leak.html Accessed 10-January-2015.

[8] D. Crockford, JavaScript: The Good Parts, 2008

[9] L. Eadicicco, Business Journal, Google's Chromebook Is Killing The iPad In One Key Market, [WWW], Available: http://uk.businessinsider.com/google-chromebooks-outsell-ipads-education-2014-12?r=US Accessed 10-January-2015.

[10] ECMAScript® 2015 Language Specification, [WWW], Available, http://www.ecma-international.org/ecma-262/6.0/ Accessed 20-October-2015.

[11] Thoughts on Specifying Garbage Collection Semantics, [WWW], Available: http://wiki.ecmascript.org/doku.php?id=strawman:gc_semantics Accessed 31-March-2015.

[12] strawman weak refs, [WWW], Available: http://wiki.ecmascript.org/doku.php?id=strawman:weak_refs Accessed 31-March-2015.

[13] Facebook, Flow, a new static type checker for JavaScript, [WWW], Available: https://code.prod.facebook.com/posts/1505962329687926/flow-a-new-static-type-checker-for-javascript/ Accessed 10-January-2015.

[14] D. Glasser, A surprising JavaScript memory leak found at Meteor, [WWW], Available: http://point.davidglasser.net/2013/06/27/surprising-javascript-memory-leak.html Accessed 10-January-2015.

[15] A. Grosskurth, M. Godfrey, A Reference Architecture for Web Browsers

[16] T. Garsiel, How Browsers Work: Behind the scenes of modern web browsers, [WWW], Available: http://taligarsiel.com/Projects/howbrowserswork1.htm Accessed 10-January-2015.

[17] Google Developers, Optimizing JavaScript, [WWW], Available: https://developers.google.com/speed/articles/optimizing-javascript Accessed 31-Mar-2015.

[18] Google Developers, Chrome v8 Design Elements, [WWW], Available: https://developers.google.com/v8/design Accessed 10-January-2015.

[19] Google Chrome, JavaScript Memory Profiling, [WWW], Available: https://developer.chrome.com/devtools/docs/javascript-memory-profiling Accessed 10-March-2015.

[20] PJ Hough, Debugging and Tuning Web Sites and Apps with F12 Developer Tools in IE11, [WWW], Available: http://blogs.msdn.com/b/ie/archive/2013/07/29/debugging-and-tuning-web-sites-and-apps-with-f12-developer-tools-in-ie11.aspx Accessed 10-January-2015.

[21] M. Jazayeri Trends In Web Applications

[22] T. Kadlec, Setting a Performance Budget, [WWW], Available: http://timkadlec.com/2013/01/setting-a-performance-budget/

[23] Konqueror, Web Browser, [WWW], Available: https://konqueror.org/features/browser.php

[24] T. Mikkonen, A. Taivalsaari, Web Applications - Spaghetti Code for the 21st century

[25] . Mozilla Developer Network, WeakMap, [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap

[26] . Mozilla Developer Network, WeakSet, [WWW], Availble: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet

[27] Mozilla Developer Network, ECMAScript 6 support in Mozilla, [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla

[28] Mozilla Developer Network, A re-introduction to JavaScript (JS tutorial), [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

[29] Mozilla Developer Network, Window, [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/API/Window

[30] Mozilla Developer Network (MDN), Strict Mode, [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

[31] Mozilla Developer Network (MDN), Data Structures [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

[32] Mozilla Developer Network, Statements and Declarations, var hoisting, [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var#var_hoisting

[33] Mozilla Developer Network, HTML5, [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5

[34] Mozilla Developer Network (MDN), Memory Management [WWW], Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

[35] Generational Garbage Collection in Firefox [WWW], Available: https://hacks.mozilla.org/2014/09/generational-garbage-collection-in-firefox/

[36] Mozilla Research, K. McAllister, JavaScript: Servo's only garbage collector, [WWW], Available: https://blog.mozilla.org/research/2014/08/26/javascript-servos-only-garbage-collector/

[37] Mozilla Manifesto [WWW], Available: https://www.mozilla.org/en-US/about/manifesto/

[38] National Center for Supercomputing Applications, NCSA Mosaic, [WWW], Available: http://www.ncsa.illinois.edu/enabling/mosaic

[39] Node.js, process, [WWWW], Available: https://nodejs.org/api/process.html

[40] S. O'Grady, The RedMonk Programming Language Rankings: January 2015, [WWW], Available: http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

[41] A. Osmani, Memory Management Masterclass [WWW], Available: https://speakerdeck.com/addyosmani/javascript-memory-management-masterclass

[42] L. Paulson, Developers shift to dynamic programming languages

[43] E. Protalinski, Opera confirms it will follow Google and ditch WebKit for Blink, as part of its commitment to Chromium, [WWW], Available: http://thenextweb.com/insider/2013/04/04/opera-confirms-it-will-follow-google-and-ditch-webkit-for-blink-as-part-of-i

[44] M. Pohja, Web Applications User Interface Technologies

[45] J. Spolsky, The Law of Leaky Abstractions, [WWW], Available: http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[46] M. Takada, Single page apps in depth, [WWW], Available: http://singlepageappbook.com/single-page.html

[47] A. Taivalsaari, T. Mikkonen, M. Anttonen, A. Salminen, The Death Of Binary Software - End User Software Moves to the Web,

[48] A. Taivalsaari, T. Mikkonen, D. Ingalls, K. Palacz, Web Browser as an Application Platform

[49] A. Taivalsaari, T. Mikkonen, Apps Vs. Open Web - The Battle of the Decade

[50] A. Taivalsaari, T. Mikkonen, Reports of Web's Death Are Greatly Exaggerated

[51] Memory Management Glossary [WWW], Available: http://www.memoryManagement.org/glossary/g.html#term-garbage-collection

[52] ESLint - The pluggable linting utility for JavaScript, [WWW], Available: http://eslint.org/

[53] Browser Usage, StatCounter [WWW], Available: https://en.wikipedia.org/wiki/File:Usage_share_of_web_browsers_(Source_StatCounter).svg

[54] Effectively Managing Memory at Gmail scale Available: http://www.html5rocks.com/en/tutorials/memory/effectivemanagement/

[55] Fling, B. Mobile Design and Development: Practical Techniques for Creating Mobile Sites and Web Apps. O'Reilly Media, Inc., 2009.

[56] Re: Browser implementations, prior to rec, used for justification, [WWW], Available: http://lists.w3.org/Archives/Public/public-html/2010Jan/0107.html

[57] D. Flanagan, JavaScript: The Definitive Guide

[58] J. Conrod, A Tour Of v8, [WWW], Available: http://www.jayconrod.com/posts/51/a-tour-of-v8-full-compiler

[59] M. Laine, D. Shestakov, E. Litvinova, P. Vuorimaa, Towards Unified Web Application Development

[60] Node.js, About Node.js [WWW], Available: http://nodejs.org/about/

[61] Source/WebCore/ChangeLog [WWW], Available: https://bugs.webkit.org/attachment.cgi?id=154876&action=prettypatch

[62] WebKit Bugzilla, JSC should have a generational garbage collector, [WWW], Available: https://bugs.webkit.org/show_bug.cgi?id=121074

[63] W3C Recommendation 08 January 2015, Indexed Database API, [WWW], Available: http://www.w3.org/TR/IndexedDB/

[64] Lynx (web browser), Wikipedia, [WWW], Available: http://en.wikipedia.org/wiki/Lynx_(web_browser)

[65] Netscape Navigator, Wikipedia, [WWW], Available: https://en.wikipedia.org/wiki/Netscape_Navigator

[66] Internet Explorer, Wikipedia, [WWW], Available: https://en.wikipedia.org/wiki/Internet_Explorer

[67] Opera (web browser), Wikipedia, [WWW], Available: https://en.wikipedia.org/wiki/Opera_%28web_browser%29

[68] Mozilla, Wikipedia, [WWW], Available: https://en.wikipedia.org/wiki/Mozilla

[69] Safari (web browser), Wikipedia, [WWW], Available: http://en.wikipedia.org/wiki/Safari_%28web_browser%29

[70] Chromium (web browser), Wikipedia, [WWW], Available: http://en.wikipedia.org/wiki/Chromium_%28web_browser%29

[71] Observer Pattern, Wikipedia, [WWW], Available: http://en.wikipedia.org/wiki/Observer_pattern

[72] Lapsed listener problem, Wikipedia, [WWW], Available: http://en.wikipedia.org/wiki/Lapsed_listener_problem

[73] Bring Your Own Device, Wikipedia, [WWW], Available: https://en.wikipedia.org/wiki/Bring_your_own_device

[74] Rich Internet Application, Wikipedia, [WWW] Available: https://en.wikipedia.org/wiki/Rich_Internet_application

[75] Resident Set Size, Wikipedia, [WWW], Available: http://en.wikipedia.org/wiki/Resident_set_size

[76] TypeScript, [WWW], Available: http://www.typescriptlang.org/

[77] R. Shahriyar, S. M. Blackburn, K. M. McKinley, Fast Concervative Garbage Collection,

[78] Advanced Memory Management Programming Guide, Apple

[79] JSLint, [WWW], Available: http://www.jslint.com/lint.html

[80] Common JS, [WWW], Available: http://www.commonjs.org/

[81] AMD, [WWW], Available: https://github.com/amdjs/amdjs-api/wiki/AMD

[82] RequireJs, [WWW], Available: http://requirejs.org/

[83] HTML, Living Standard - Last Updated 29 January 2015 [WWW], Available: https://html.spec.whatwg.org/multipage/introduction.html#introduction

[84] Web Components Current Status, [WWW], Available, http://www.w3.org/standards/techs/components#w3c_all

[85] Shadow DOM, W3C Editor's Draft 23 January 2015, [WWW], Available, http://w3c.github.io/webcomponents/spec/shadow/

[86] Web Performance Working Group, [WWW], Available: http://www.w3.org/2010/webperf/

[87] Rich Web Client Activity Statement, [WWW], Available: http://www.w3.org/2006/rwc/Activity.html

[88] Document Object Model, [WWW], Available: http://www.w3.org/DOM/

[89] Document Object Model Core, Fundamental Interfaces, Node, [WWW], Available: http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1950641247

[90] The WebKit Open Source Project [WWW], Available: http://www.webkit.org/